# Computation-Centric Memory Models

Matteo Frigo[*]

MIT Laboratory for Computer Science
545 Technology Square NE43-203
Cambridge, MA 02139
athena@theory.lcs.mit.edu

Victor Luchangco[†]

Massachusetts Institute of Technology
545 Technology Square NE43-369
Cambridge, MA 02139
victor_l@theory.lcs.mit.edu

## Abstract

We present a ***computation-centric theory*** of memory models. Unlike traditional processor-centric models, computation-centric models focus on the logical dependencies among instructions rather than the processor that happens to execute them. This theory allows us to define what a memory model is, and to investigate abstract properties of memory models. In particular, we focus on ***constructibility***, which is a necessary property of those models that can be implemented exactly by an online algorithm. For a nonconstructible model, we show that there is a natural way to define the ***constructible version*** of that model. We explore the implications of constructibility in the context of ***dag-consistent memory models***, which do not require that memory locations be serialized. The strongest dag-consistent model, called NN-dag consistency, is not constructible. However, its constructible version is equivalent to a model that we call ***location consistency***, in which each location is serialized independently.

## 1 Introduction

A memory model specifies the values that may be returned by the memory of a computer system in response to instructions issued by a program. In this paper, we develop a ***computation-centric theory*** of memory models in which we can reason about memory models abstractly. We define formally what a memory model is, and we investigate the implications of ***constructibility***, an abstract property which is necessary for a model to be maintainable exactly by an online algorithm. The computation-centric theory is based on the two concepts of a ***computation*** and an ***observer function***.

Most existing memory models [DSB86, AH90, Goo89, GLL$^+$90, KCZ92, BZS93, ISL96] are expressed in terms of *processors* acting on *memory*. We call these memory models ***processor-centric***; the memory model specifies what happens when a processor performs some action on memory. In contrast, the philosophy of this paper is to separate the logical dependencies among instructions (the computation) from the way instructions are mapped to processors (the schedule). For example, in a multithreaded program, the programmer specifies several execution threads and certain dependencies among the threads, and expects the behavior of the program to be specified independently of which processor happens to execute a particular thread. Computation-centric memory models focus on the computation alone, and not on the schedule. While the processor-centric description has the advantage of modeling real hardware closely, our approach allows us to define formal properties of memory models that are independent of any implementation.

A ***computation*** is an abstraction of a parallel instruction stream. The computation specifies machine instructions and dependencies among them. A computation does not model a parallel program, but rather the way a program unfolds in a particular execution. (A program may unfold in different ways because of input values and nondeterministic or random choices.) We model the result of this unfolding process by a directed acyclic graph whose nodes represent instances of instructions in the execution. For example, a computation could be generated using a multithreaded language with fork/join parallelism (such as Cilk [BJK$^+$95]). Computations are by no means limited to modeling multithreaded programs, however. In this paper, we assume that the computation is given, and defer the important problem of determining which computations a given program generates. We can view computations as providing a means for *post mortem* analysis, to verify whether a system meets a specification by checking its behavior after it has finished executing.

[*]Supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-94-1-0985, and by a Digital Equipment Corporation Fellowship.

[†]Supported by AFOSR-ONR contract F49640-94-1-0199, by AFOSR contract F49640-97-1-0337, by ARPA contracts N00014-92-J-4033 and F19628-95-C-0118, and by NSF grant 9225124-CCR.

To specify memory semantics, we use the notion of an ***observer function*** for a computation. Informally, for each node of the computation (i.e., an instance of an instruction) that reads a value from the memory, the observer function specifies the node that wrote the value that the read operation receives. Computation-centric memory models are defined by specifying a set of valid observer functions for each computation. A memory implements a memory model if, for every computation, it always generates an observer function belonging to the model.

Within the computation-centric theory, we define a property we call ***constructibility***. Informally, a nonconstructible memory model cannot be implemented exactly by an online algorithm; any online implementation of a nonconstructible memory must maintain a strictly stronger constructible model. We find constructibility interesting because it makes little sense to adopt a memory model if any implementation of it must maintain a stronger model. One important result of this paper is that such a stronger model is unique. We prove that for any memory model $\Delta$, the class of constructible memory models stronger than $\Delta$ has a unique weakest element, which we call the ***constructible version*** $\Delta^*$ of $\Delta$.

We discuss two approaches for specifying memory models within this theory. In the first approach, a memory model is defined in terms of topological sorts of the computation. Using this approach, we generalize the definition of ***sequential consistency*** [Lam79], and define ***location consistency***,[1] a memory model in which every location is serialized independently of other locations. In the second approach, a memory model is defined by imposing certain constraints on the value that the observer function can assume on paths in the computation dag. Using this approach, we explore the class of ***dag-consistent*** memory models, a generalization of the ***dag consistency*** of [BFJ$^+$96b, BFJ$^+$96a, Joe96]. Such models do not even require that a single location be serialized, and are thus strictly weaker than the other class of models. Nonetheless, we found an interesting link between location consistency, dag consistency and constructibility. The strongest variant of dag consistency (called ***NN-dag consistency***) is not constructible, and is strictly weaker than location consistency. Its constructible version, however, turns out to be the same model as location consistency.

We believe that the advantages of the computation-centric framework transcend the particular results mentioned so far. First, we believe that reasoning about computations is easier than reasoning about processors. Second, the framework is completely formal, and thus we can make rigorous proofs of the correctness of a memory. Third, our approach allows us to generalize familiar memory models, such as sequential consistency. Most of the simplicity of our theory comes from ignoring the fundamental issue of how programs generate computations. This simplification does not come without cost, however. The computation generated by a program may depend on the values received from the memory, which in turn depend on the computation. It remains important to account for this circularity within a unified theory. We believe, however, that the problem of memory semantics alone is sufficiently difficult that it is better to isolate it initially.

The rest of this paper is organized as follows. In Section 2, we present the basic computation-centric theory axiomatically. In Section 3, we define constructibility, prove the uniqueness of the constructible version, and establish necessary and sufficient conditions for constructibility to hold. In Section 4, we discuss models based on a topological sort, and give computation-centric definitions of sequential consistency [Lam79] and location consistency. In Section 5, we define the class of dag-consistent memory models and investigate the relations among them. In Section 6, we prove that location consistency is the constructible version of NN-dag consistency. Finally, we situate our work in the context of related research in Section 7.

## 2 Computation-centric memory models

In this section, we define the basic concepts of the computation-centric theory of memory models. The main definitions are those of a ***computation*** (Definition 1), an ***observer function*** (Definition 2), and a ***memory model*** (Definition 3). We also define two straightforward properties of memory models called ***completeness*** and ***monotonicity***.

We start with a formal definition of memory. A ***memory*** is characterized by a set $\mathcal{L}$ of ***locations***, a set $\mathcal{O}$ of abstract instructions (such as read and write), and a set of ***values*** that can be stored at each location. In the rest of the paper, we abstract away the actual data, and consider a memory to be characterized by $\mathcal{L}$ and $\mathcal{O}$, using values only for concrete examples.

For a set $\mathcal{O}$ of abstract instructions, we formally define a computation as follows.

**Definition 1** A ***computation*** $C = (G, \mathit{op})$ is a pair of a finite directed acyclic graph (dag) $G = (V, E)$ and a function $\mathit{op} : V \mapsto \mathcal{O}$.

For a computation $C$, we use $G_C$, $V_C$, $E_C$ and $\mathit{op}_C$ to indicate its various components. The smallest computation is the ***empty computation*** $\varepsilon$, which has an empty dag. Intuitively, each node $u \in V$ represents an instance of the instruction $\mathit{op}(u)$, and each edge indicates a dependency between its endpoints.

The way a computation is generated from an actual execution depends on the language used to write the program. For example, consider a program written in a language with fork/join parallelism. The execution of the program can be viewed as a set of operations on memory that obey the dependencies imposed by the fork/join constructs. Although

[1]Location consistency is often called coherence in the literature [HP96]. It is *not* the model with the same name introduced by Gao and Sarkar [GS95]. See [Fri98] for a justification of this terminology.

the issues of how the computation is expressed and scheduled are extremely important, they are outside the scope of this paper. The reader is referred to [Blu95, Joe96] for one way to deal with these issues. In this paper, we consider the computation as fixed and given *a priori*.

In this paper, we consider only read-write memories. We denote reads and writes to location $l$ by $R(l)$ and $W(l)$ respectively. For the rest of the paper, the set of instructions is assumed to be $\mathcal{O} = \{R(l) : l \in \mathcal{L}\} \cup \{W(l) : l \in \mathcal{L}\} \cup \{N\}$, where $N$ denotes any instruction that does not access the memory (a "no-op").

We now define some terminology for dags and computations. If there is a path from node $u$ to node $v$ in the dag $G$, we say that $u$ ***precedes*** $v$ in $G$, and we write $u \preceq_G v$. We may omit the dag and write $u \preceq v$ when it is clear from context. We often need to indicate strict precedence, in which case we write $u \prec v$. A ***relaxation*** of a dag $G = (V, E)$ is any dag $(V, E')$ such that $E' \subseteq E$. A ***prefix*** of $G$ is any subgraph $G' = (V', E')$ of $G$ such that if $(u, v) \in E$ and $v \in V'$, then $u \in V'$ and $(u, v) \in E'$.

A ***topological sort*** $T$ of $G = (V, E)$ is a total order on $V$ consistent with the precedence relation, i.e., $u \preceq_G v$ implies that $u$ precedes $v$ in $T$. The precedence relation of the topological sort is denoted with $u \preceq_T v$. We represent topological sorts as sequences, and denote by $\mathcal{TS}(G)$ the set of all topological sorts of a dag $G$. Note that for any $V' \subseteq V$, if $G'$ is the subgraph of $G$ induced by $V'$ and $G''$ is the subgraph induced by $V - V'$, and $T'$ and $T''$ are topological sorts of $G'$ and $G''$ respectively, then the concatenation of $T'$ and $T''$ is a topological sort of $G$ if and only if for all $u \in V'$ and $v \in V - V'$, we have $v \not\preceq_G u$.

For a computation $C = (G, op)$, if $G'$ is a subgraph of $G$ and $op'$ is the restriction of $op$ to $G'$, then $C' = (G', op')$ is a ***subcomputation*** of $C$. We also call $op'$ the ***restriction*** of $op$ to $C'$, and denote it by $op|_{C'}$, i.e., $op|_{C'}(u) = op(u)$ for all $u \in V_{C'}$. We abuse notation by using the same terminology for computations as for dags. For example, $C'$ is a ***prefix*** of $C$ if $G_{C'}$ is a prefix of $G_C$ and $op_{C'} = op_C|_{C'}$. Similarly, $\mathcal{TS}(C) = \mathcal{TS}(G_C)$. In addition, $C$ is an ***extension*** of $C'$ by $o \in \mathcal{O}$ if $C'$ is a prefix of $C$, $V_C = V_{C'} \cup \{u\}$ for some $u \notin V_{C'}$ and $op_C(u) = o$. Note that if $C'$ is a prefix of $C$ with $|V_C| = |V_{C'}| + 1$ then $C$ is an extension of $C'$ by $op_C(u)$, where $u \in V_C - V_{C'}$.

We imagine a computation as being executed in some way by one or more processors, subject to the dependency constraints specified by the dag, and we want to define precisely the semantics of the read and write operations. For this purpose, rather than specifying the meaning of read and write operations directly, we introduce a technical device called an ***observer function***. For every node $u$ in the computation and for every location $l$, the value of the observer function $v = \Phi(l, u)$ is another node that writes to $l$. The idea is that $u$ "observes" the write performed by $v$, so that if $u$ reads $l$, it receives the value written by $v$. The observer function can assume the special value $\perp$, indicating that no write has been observed, in which case a read operation receives an undefined value. Note that $\perp$ is not a value stored at a location, but an element of the range of the observer function similar to a node of the computation. For notational convenience, we extend the precedence relation so that $\perp \prec u$ for every node $u$ of any computation, and we also include $\perp$ as a node in the domain of observer functions.

**Definition 2** An ***observer function*** for a computation $C$ is a function $\Phi : \mathcal{L} \times V_C \cup \{\perp\} \mapsto V_C \cup \{\perp\}$ satisfying the following properties for all $l \in \mathcal{L}$ and $u \in V_C \cup \{\perp\}$:

2.1. If $\Phi(l, u) = v \neq \perp$ then $op_C(v) = W(l)$.

2.2. $u \not\prec \Phi(l, u)$.

2.3. If $u \neq \perp$ and $op_C(u) = W(l)$ then $\Phi(l, u) = u$.

Informally, every observed node must be a write (part 2.1), and a node cannot precede the node it observes (part 2.2). Furthermore, every write must observe itself (part 2.3). Note that Condition 2.2 implies $\Phi(l, \perp) = \perp$ for all $l \in \mathcal{L}$. The empty computation has a unique observer function, which we denote by $\Phi_\varepsilon$.

The observer function allows us to abstract away from memory values, and to give memory semantics even to nodes that do not perform memory operations. In other words, our formalism may distinguish two observer functions that produce the same execution. We choose this formalism because it allows a computation node to denote some form of synchronization, which affects the memory semantics even if the node does not access the memory.

A ***memory model*** $\Delta$ is a set of pairs of computations and observer functions, including the empty computation and its observer function,[2] as stated formally by the next definition.

**Definition 3** A ***memory model*** is a set $\Delta$ such that

$$\{(\varepsilon, \Phi_\varepsilon)\} \subseteq \Delta \subseteq \{(C, \Phi) : \Phi \text{ is an observer function for } C\}$$

The next definition is used to compare memory models.

**Definition 4** A model $\Delta$ is ***stronger*** than a model $\Delta'$ if $\Delta \subseteq \Delta'$. We also say that $\Delta'$ is ***weaker*** than $\Delta$.

Notice that the subset, not the superset, is said to be stronger, because the subset allows fewer memory behaviors.

A memory model may provide an observer function only for some computations. It is natural to restrict ourselves to those models that define at least one observer function for each computation. We call such models complete. Formally, a memory model $\Delta$ is ***complete*** if, for every computation $C$, there exists an observer function $\Phi$ such that $(C, \Phi) \in \Delta$.

From the definitions of weaker and complete, it follows that any model weaker than some complete model is also

[2]This is a technical requirement to simplify boundary cases.

complete. Formally, if $\Delta$ is complete and $\Delta' \supseteq \Delta$, then $\Delta'$ is also complete.

Another natural property for memory models to satisfy is that relaxations of a computation should not invalidate observer functions for the original computation. We call this property monotonicity.

**Definition 5** A memory model $\Delta$ is ***monotonic*** if for all $(C, \Phi) \in \Delta$, we also have $(C', \Phi) \in \Delta$, for all relaxations $C'$ of $C$.

Monotonicity is a technical property that simplifies certain proofs (for example, see Theorem 12), and we regard it as a natural requirement for any "reasonable" memory model.

## 3 Constructibility

In this section, we define a key property of memory models that we call ***constructibility***. Constructibility says that if we have a computation and an observer function in some model, it is always possible to extend the observer function to a "bigger" computation. Not all memory models are constructible. However, there is a natural way to define a unique ***constructible version*** of a nonconstructible memory model. At the end of the section, we give a necessary and sufficient condition for the constructibility of monotonic memory models.

The motivation behind constructibility is the following. Suppose that, instead of being given completely at the beginning of an execution, a computation is revealed one node at a time by an adversary.[3] Suppose also that there is an algorithm that maintains a given memory model online. Intuitively, the algorithm constructs an observer function as the computation is revealed. Suppose there is some observer function for the part of the computation revealed so far, but when the adversary reveals the next node, there is no way to assign a value to it that satisfies the memory model. In this case, the consistency algorithm is "stuck". It should have chosen a different observer function in the past, but that would have required some knowledge of the future. Constructibility says that this situation cannot happen: if $\Phi$ is a valid observer function in a constructible model, then there is always a way to extend $\Phi$ to a "bigger" computation as it is revealed.

**Definition 6** A memory model $\Delta$ is ***constructible*** if the following property holds: for all computations $C'$ and for all prefixes $C$ of $C'$, if $(C, \Phi) \in \Delta$ then there exists an observer function $\Phi'$ for $C'$ such that $(C', \Phi') \in \Delta$ and the restriction of $\Phi'$ to $C$ is $\Phi$, i.e., $\Phi'|_C = \Phi$.

Completeness follows immediately from constructibility, since the empty computation is a prefix of all computations and, together with its unique observer function, belongs to every memory model.

Not all memory models are constructible; we shall discuss some nonconstructible memory models in Section 5. However, a nonconstructible model $\Delta$ can be strengthened in an essentially unique way until it becomes constructible. More precisely, the set of constructible models stronger than $\Delta$ contains a unique weakest element $\Delta^*$, which we call the ***constructible version*** of $\Delta$. To prove this statement, we first prove that the union of constructible models is constructible.

**Lemma 7** Let $\mathcal{S}$ be a (possibly infinite) set of constructible memory models. Then $\bigcup_{\Delta \in \mathcal{S}} \Delta$ is constructible.

*Proof:* Let $C'$ be a computation and $C$ be a prefix of $C'$. We must prove that, if $(C, \Phi) \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, then an extension $\Phi'$ of the observer function $\Phi$ exists such that $(C', \Phi') \in \bigcup_{\Delta \in \mathcal{S}} \Delta$.

If $(C, \Phi) \in \bigcup_{\Delta \in \mathcal{S}} \Delta$ then $(C, \Phi) \in \Delta$ for some $\Delta \in \mathcal{S}$. Since $\Delta$ is constructible, there exists an observer function $\Phi'$ for $C'$ such that $(C', \Phi') \in \Delta$ and $\Phi'|_C = \Phi'$. Thus, $(C', \Phi') \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, as required. ■

We now define the constructible version of a model $\Delta$, and prove that it is the weakest constructible model stronger than $\Delta$.

**Definition 8** The ***constructible version*** $\Delta^*$ of a memory model $\Delta$ is the union of all constructible models stronger than $\Delta$.

**Theorem 9** For any memory model $\Delta$,

9.1. $\Delta^* \subseteq \Delta$;

9.2. $\Delta^*$ is constructible;

9.3. for any constructible model $\Delta'$ such that $\Delta' \subseteq \Delta$, we have $\Delta' \subseteq \Delta^*$.

*Proof:* $\Delta^*$ satisfies Conditions 9.1 and 9.3 by construction, and Condition 9.2 because of Lemma 7. ■

In two theorems, we establish conditions that guarantee constructibility. Theorem 10 gives a sufficient condition for the constructibility of general memory models. For monotonic memory models, the condition is simpler (Theorem 12).

**Theorem 10** A memory model $\Delta$ is constructible if for any $(C, \Phi) \in \Delta$, $o \in \mathcal{O}$, and extension $C'$ of $C$ by $o$, there exists an observer function $\Phi'$ for $C'$ such that $(C', \Phi') \in \Delta$ and $\Phi = \Phi'|_C$.

[3] This is the case with multithreaded languages, such as Cilk [Blu95, Joe96].

*Proof:* We must prove that if $C$ is a prefix of $C'$ and $(C, \Phi) \in \Delta$, then there exists an observer function $\Phi'$ for $C'$ such that $(C', \Phi') \in \Delta$ and $\Phi'|_C = \Phi$.

Since $C$ is a prefix of $C'$, there exists a sequence of computations $C_0, C_1, \ldots, C_k$ such that $C_0 = C$, $C_k = C'$, and $C_i$ is an extension of $C_{i-1}$ by some $o_i \in \mathcal{O}$ for all $i = 1, \ldots, k$, where $k = |V_{C'}| - |V_C|$.

The proof of the theorem is by induction on $k$. The base case $k = 0$ is trivial since $C' = C$. Now, suppose inductively that there exists $\Phi_{k-1}$ such that $(C_{k-1}, \Phi_{k-1}) \in \Delta$. Since $C'$ is an extension of $C_{k-1}$ by $o_k$, the theorem hypothesis implies that an observer function $\Phi'$ exists such that $(C', \Phi') \in \Delta$, as required to complete the inductive step. ■

For monotonic memory models, we do not need to check every extension of a computation to prove constructibility, but rather only a small class of them, which we call the ***augmented computations***. An augmented computation is an extension by one "new" node, where the "new" node is a successor of all "old" nodes.

**Definition 11** Let $C$ be a computation and $o \in \mathcal{O}$ be any operation. The ***augmented computation*** of $C$ by $o$, denoted $\mathit{aug}_o(C)$, is the computation $C'$ such that

$$\begin{aligned} V_{C'} &= V_C \cup \{\mathit{final}(C)\} \\ E_{C'} &= E_C \cup \{(v, \mathit{final}(C)) : v \in V_C\} \\ \mathit{op}_{C'}(v) &= \begin{cases} \mathit{op}_C(v) & \text{for } v \in V_C \\ o & \text{for } v = \mathit{final}(C) \end{cases}, \end{aligned}$$

where $\mathit{final}(C) \notin V_C$ is a new node.

The final theorem of this section states that if a monotonic memory model can extend the observer function for any computation to its augmented computations, then the memory model is constructible.

**Theorem 12** A monotonic memory model $\Delta$ is constructible if and only if for all $(C, \Phi) \in \Delta$ and $o \in \mathcal{O}$, there exists an observer function $\Phi'$ such that $(\mathit{aug}_o(C), \Phi') \in \Delta$ and $\Phi'|_C = \Phi$.

*Proof:* The "$\Rightarrow$" part is obvious, since $C$ is a prefix of $\mathit{aug}_o(C)$.

For the "$\Leftarrow$" direction, suppose $(C, \Phi) \in \Delta$ and $o \in \mathcal{O}$. By hypothesis, there exists $\Phi'$ such that $(\mathit{aug}_o(C), \Phi') \in \Delta$. For any extension $C'$ of $C$ by $o$, note that $C'$ is a relaxation of $\mathit{aug}_o(C)$. Since $\Delta$ is monotonic, we also have $(C', \Phi') \in \Delta$. Thus, by Theorem 10, $\Delta$ is constructible. ■

One interpretation of Theorem 12 is the following. Consider an execution of a computation. At any point in time some prefix of the computation will have been executed. If at all times it is possible to define a "final" state of the memory (given by the observer function on the final node of the augmented computation) then the memory model is constructible.

## 4 Models based on topological sorts

In this section, we define two well known memory models in terms of topological sorts of a computation. The first model is ***sequential consistency*** [Lam79]. The second model is sometimes called ***coherence*** in the literature [GS95, HP96]; we call it ***location consistency***. Both models are complete, monotonic and constructible. Because we define these models using computations, our definitions generalize traditional processor-centric ones without requiring explicit synchronization operations.

It is convenient to state both definitions in terms of the "last writer preceding a given node", which is well defined if we superimpose a total order on a computation, producing a topological sort.

**Definition 13** Let $C$ be a computation, and $T \in \mathcal{TS}(C)$ be a topological sort of $C$. The ***last writer function*** according to $T$ is $\mathcal{W}_T : \mathcal{L} \times V_C \cup \{\bot\} \mapsto V_C \cup \{\bot\}$ such that for all $l \in \mathcal{L}$ and $u \in V_C \cup \{\bot\}$:

13.1. If $\mathcal{W}_T(l, u) = v \neq \bot$ then $\mathit{op}_C(v) = W(l)$.

13.2. $\mathcal{W}_T(l, u) \preceq_T u$.

13.3. $\mathcal{W}_T(l, u) \prec_T v \preceq_T u \implies \mathit{op}_C(v) \neq W(l)$ for all $v \in V_C$.

We state without proof two straightforward facts about last writer functions. The first states that Definition 13 is well defined. The second states that if $w$ is the last writer preceding a node $u$, then it is also the last writer preceding any node between $w$ and $u$.

**Theorem 14** For any topological sort $T$, there exists a unique last writer function according to $T$.

*Proof:* Omitted. ■

**Theorem 15** For any computation $C$, if $\mathcal{W}_T$ is the last writer function according to $T$ for some $T \in \mathcal{TS}(C)$ then for all $u, v \in V_C$ and $l \in \mathcal{L}$ such that $\mathcal{W}_T(l, u) \prec_T v \preceq_T u$, we have $\mathcal{W}_T(l, v) = \mathcal{W}_T(l, u)$.

*Proof:* Omitted. ■

We use the last writer function for defining memory models, which is possible because the the last writer function is an observer function, as stated in the next theorem.

**Theorem 16** Let $C$ be a computation, and $T \in \mathcal{TS}(C)$ be a topological sort of $C$. The last writer function $\mathcal{W}_T$ is an observer function for $C$.

*Proof:* Condition 13.1 is the same as Condition 2.1 and Condition 2.2 is implied by Condition 13.2. Finally, note that the contrapositive of Condition 13.3 with $v = u \neq \bot$ is $op_C(u) = W(l) \implies \mathcal{W}_T(l,u) \not\prec_T u$. Using Condition 13.2, this simplifies to $op_C(u) = W(l) \implies \mathcal{W}_T(l,u) = u$, thus proving Condition 2.3. ■

We define sequential consistency using last writer functions.

**Definition 17** ***Sequential consistency*** is the memory model

$$\mathrm{SC} = \{(C, \mathcal{W}_T) : T \in \mathcal{TS}(C)\}$$

This definition captures the spirit of Lamport's original model [Lam79], that there exists a global total order of events observed by all nodes. However, unlike Lamport's definition, it does not restrict dependencies to be sequences of operations at each processor, nor does it depend on how the computation is mapped onto processors.

Sequential consistency requires that the topological sort be the same for all locations. By allowing a different topological sort for each location, we define a memory model that is often called ***coherence*** [GS95, HP96]. We believe that a more appropriate name for this model is ***location consistency***, even though the same name is used in [GS95] for a different memory model.[4]

**Definition 18** ***Location consistency*** is the memory model

$$\mathrm{LC} = \{(C, \Phi) : \forall l\ \exists T_l \in \mathcal{TS}(C)\ \forall u,\ \Phi(l,u) = \mathcal{W}_{T_l}(l,u)\}$$

Location consistency requires that all writes to the same location behave *as if* they were serialized. This need not be the case in the actual implementation. For example, the BACKER algorithm from [BFJ$^+$96b, BFJ$^+$96a] maintains location consistency, even though it may keep several incoherent copies of the same location. In Section 6, we prove that location consistency is the constructible version of a model we call NN-dag consistency.

It follows immediately from the definitions that SC is stronger than LC. In fact, this relation is strict as long as there is more than one location.

Both SC and LC are complete memory models, because an observer function can be constructed for any computation by sorting the dag and using the last writer function. We now prove that they are also monotonic and constructible.

**Theorem 19** SC and LC are monotonic and constructible memory models.

*Proof:* The monotonicity of both follows immediately from the definition since $\mathcal{TS}(C) \subseteq \mathcal{TS}(C')$ for all relaxations $C'$ of $C$.

[4]See [Fri98] for a discussion of this terminology.

For constructibility, we give only the proof for SC; the proof for LC is similar. Since SC is monotonic, we only need to prove that it is possible to extend any observer function for a computation to its augmented computation, and then apply Theorem 12.

If $(C, \Phi) \in \mathrm{SC}$ then, by definition of SC, $\Phi = \mathcal{W}_T$ for some topological sort $T \in \mathcal{TS}(C)$. For each $o \in \mathcal{O}$, consider the augmented computation $\mathit{aug}_o(C)$, and let $T'$ be the following total order of the nodes of $\mathit{aug}_o(C)$: all the nodes of $C$ in $T$ order, followed by $\mathit{final}(C)$. It is immediate that $T'$ is a topological sort of $\mathit{aug}_o(C)$. Thus, $\mathcal{W}_{T'}$ is a valid SC observer function for $\mathit{aug}_o(C)$, and $\mathcal{W}_{T'}|_C = \mathcal{W}_T = \Phi$. The conclusion follows by application of Theorem 12. ■

## 5 Dag-consistent memory models

In this section, we consider the class of ***dag-consistent*** memory models, which are not based on topological sorts of the computation. Rather, dag-consistent models impose conditions on the value that the observer function can assume on paths in the computation. We focus on four "interesting" dag-consistent memory models, and investigate their mutual relations.

In the dag-consistent models the observer function obeys a restriction of the following form: If a node lies on a path between two other nodes, and the observer function assumes the value $x$ at the two end nodes, and the three nodes satisfy certain additional conditions, then the observer function also assumes the value $x$ at the middle node. The various dag consistency models differ in the additional conditions they impose on the nodes.

**Definition 20** Let $Q$ be a predicate on $\mathcal{L} \times V \times V \times V$, where $V$ is the set of all nodes of a computation. The $Q$-***dag consistency*** memory model is the set of all pairs $(C, \Phi)$ such that $\Phi$ is an observer function for $C$ and the following condition holds:

20.1. For all locations $l \in \mathcal{L}$ and nodes $u, v, w \in V_C \cup \{\bot\}$ such that $u \prec v \prec w$ and $Q(l, u, v, w)$, we have $\Phi(l,u) = \Phi(l,w) \implies \Phi(l,v) = \Phi(l,u)$.

Definition 20 is a generalization of the two definitions of dag consistency that the Cilk group of MIT (including one of the authors of the present paper) proposed in the past [BFJ$^+$96b, BFJ$^+$96a]. Varying the predicate $Q$ in Condition 20.1 yields different memory models. Note that strengthening $Q$ weakens the memory model.

In the rest of the paper, we consider four specific predicates, NN, NW, WN and WW, and the dag consistency models they define. These predicates do not depend on $w$, but only on whether $u$ and $v$ write to $l$. The rationale behind the names is that "W" stands for "write", and "N" stands for

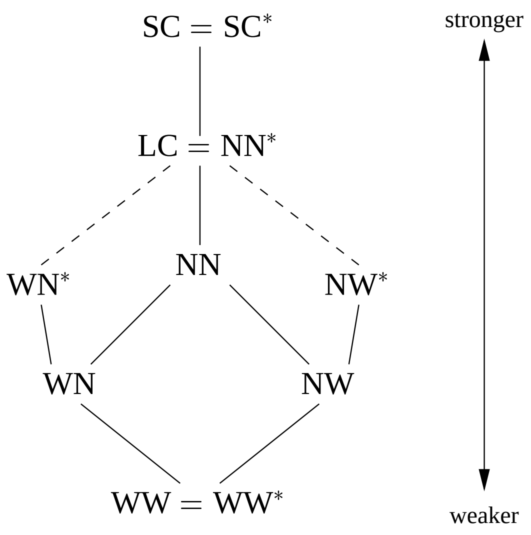

**Figure 1**: The relations among (some) dag-consistent models. A straight line indicates that the model at the lower end of the line is strictly weaker than the model at the upper end. For example, LC is strictly weaker than SC. It is known that LC $\subseteq$ WN* and that LC $\subseteq$ NW*, but we do not know whether these inclusions are strict. This situation is indicated with a dashed line.

"do not care". For example, WN means that the first node is a write and we do not care about the second. Formally,

$$\begin{aligned} \mathrm{NN}(l,u,v,w) &= \textit{true} \\ \mathrm{NW}(l,u,v,w) &= \text{“}\mathit{op}_C(v) = W(l)\text{”} \\ \mathrm{WN}(l,u,v,w) &= \text{“}\mathit{op}_C(u) = W(l)\text{”} \\ \mathrm{WW}(l,u,v,w) &= \mathrm{NW}(l,u,v,w) \wedge \mathrm{WN}(l,u,v,w) \end{aligned}$$

We use NN as a shorthand for NN-dag consistency, and similarly for WN, NW and WW.

The relations among NN, WN, NW, WW, LC and SC are shown in Figure 1. WW is the original dag consistency model defined in [BFJ+96b, Joe96]. WN is the model called dag consistency in [BFJ+96a], strengthened to avoid anomalies such as the one illustrated in Figure 2. NN is the strongest dag-consistent memory model (as proven in Theorem 21 below). Symmetry suggests that we also consider NW.

**Theorem 21** NN $\subseteq$ $Q$-dag consistency for any predicate $Q$.

*Proof:* The proof is immediate from the definition: an observer function satisfying Condition 20.1 with $Q(l,u,v,w) = \textit{true}$ will satisfy Condition 20.1 for any other predicate $Q$. ■

The rest of the paper is mostly concerned with the proof of the relations shown in Figure 1. We have already observed in Section 4 that SC is strictly stronger than LC. In the rest of this section, we give informal proofs of the relations among

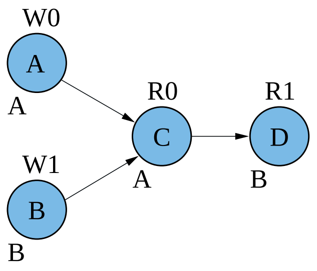

**Figure 2**: An example of a computation/observer function pair in WW and NW but not WN or NN. The computation has four nodes, A, B, C and D (the name of the node is shown inside the node). The memory consists of a single location, which is implicit. Every node performs a read or a write operation on the location, and this is indicated above the node. For example, W0 means that the node writes a 0 to the location, and R1 means that it reads a 1. The value of the observer function is displayed below each node. For example, the value of the function for node C is A, which accounts for the fact that node C reads the value written by node A.

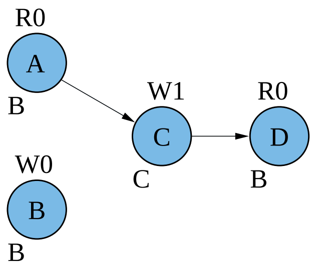

**Figure 3**: An example of a computation/observer function pair in WW and WN but not NW or NN. The conventions used in this figure are explained in Figure 2.

the dag-consistent models. Proving relations between the dag-consistent models and the models based on topological sorts, however, is more involved, and we postpone the proof that LC $\subsetneq$ NN and that LC = NN* until Section 6.

That NN $\subseteq$ NW $\subseteq$ WW and NN $\subseteq$ WN $\subseteq$ WW follows immediately from the definitions of these models. To see that these inclusions are strict and that WN $\not\subseteq$ NW and NW $\not\subseteq$ WN, consider the computation/observer function pairs shown in Figures 2 and 3. These examples illustrate operations on a single memory location, which is implicit. It is easy to verify that the first pair is in WW and NW but not WN and NN, and the second is in WW and WN but not NW and NN. We could also show that NN $\subsetneq$ NW $\cap$ WN and WW $\supsetneq$ NW $\cup$ WN, using similar examples.

To see that NN is not contructible, let $C'$ be the computation in Figure 4, and $(C, \Phi)$ be the computation/observer function pair to the left of the dashed line. It is easy to verify that $C$ is a prefix of $C'$ and that $(C, \Phi) \in$ NN. However, unless $F$ writes to the memory location, there is no way to extend $\Phi$ to $C'$ without violating NN-dag consistency. Formally, there is no $\Phi'$ such that $(C', \Phi') \in$ NN and $\Phi'|_C = \Phi$.

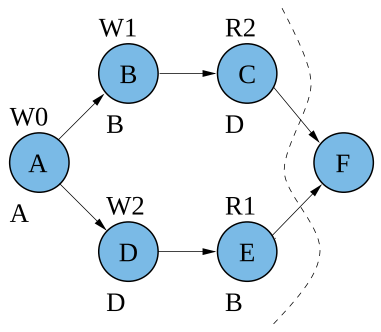

**Figure 4**: An example demonstrating the nonconstructibility of NN. The conventions used in this figure are explained in Figure 2. A new node F has been revealed by the adversary after the left part of the computation has been executed. It is not possible to assign a value to the observer function for node F satisfying NN-dag consistency.

Informally, suppose that we use an algorithm that claims to support NN-dag consistency. The adversary reveals the computation $C$, and our algorithm produces the observer function $\Phi$, which satisfies NN-dag consistency. Then the adversary reveals the new node $F$. The algorithm is "stuck"; it cannot assign a value to the observer function for $F$ that satisfies NN-dag consistency.

The same example shows that WN is not constructible, and a similar one can be used to show that NW is not constructible. WW is constructible, although we do not prove this fact in this paper.

Historically, we investigated the various dag-consistent models after discovering the problem with WN illustrated in Figure 4. Our attempts to find a "better" definition of dag consistency led us to the notion of constructibility. As Figure 1 shows, among the four models only WW is constructible. A full discussion of these models (including a criticism of WW) can be found in [Fri98]. At this stage of our research, little is known about WN$^*$ and NW$^*$, which would be alternative ways of defining dag consistency.

## 6 Dag consistency and location consistency

In this section, we investigate the relation between NN-dag consistency and location consistency. We show that location consistency is strictly stronger than any dag-consistent model, and moreover, that it is the constructible version of NN-dag consistency, i.e., LC = NN$^*$.

We begin by proving that LC is strictly stronger than NN, which implies that NN$^*$ is no stronger than LC, since LC is constructible.

**Theorem 22** LC $\subsetneq$ NN.

*Proof:* We first prove that LC $\subseteq$ NN. Let $(C, \Phi) \in$ LC. We want to prove that $(C, \Phi) \in$ NN. For each location $l$, we argue as follows: By the definition of LC, there exists $T \in \mathcal{TS}(C)$ such that $\mathcal{W}_T(l,u) = \Phi(l,u)$ for all $u \in V$.

Suppose that $u \prec v \prec w$ and $\Phi(l,u) = \Phi(l,w)$. Then $\mathcal{W}_T(l,w) = \mathcal{W}_T(l,u) \preceq_T u \prec_T v \prec_T w$. So by Theorem 15, $\mathcal{W}_T(l,v) = \mathcal{W}_T(l,u)$. Thus $\Phi(l,v) = \Phi(l,u)$ as required.

To complete the proof, we only need to note that LC $\neq$ NN since LC is constructible and NN is not. ■

From Theorems 21 and 22, it immediately follows that LC is strictly stronger than any dag-consistent memory model. And since LC is complete, it follows from that all dag-consistent models are complete.

Finally, we prove that the constructible version of NN-dag consistency is exactly location consistency.

**Theorem 23** LC = NN$^*$.

*Proof:* We first prove that NN$^* \supseteq$ LC, and then that NN$^* \subseteq$ LC. By Theorem 22, LC $\subseteq$ NN, and by Theorem 19, LC is constructible. Therefore, by Condition 9.3, we have that NN$^* \supseteq$ LC. That NN$^* \subseteq$ LC is implied by the claim that follows.

*Claim:* For any nonnegative integer $k$, suppose $(C, \Phi) \in$ NN$^*$ and $|V_C| = k$. Then for each $l \in \mathcal{L}$, there exists $T \in \mathcal{TS}(C)$ such that $\Phi(l,u) = \mathcal{W}_T(l,u)$, for all $u \in V_C$.

*Proof of claim:* The proof is by strong induction on $k$. The claim is trivially true if $k = 0$, since $C = \varepsilon$ and $\Phi = \Phi_\varepsilon$ in this case.

If $k > 0$, assume inductively that the claim is true for all computations with fewer than $k$ nodes. We prove it is true for $C$. Since NN$^*$ is constructible, Theorem 12 implies that there exists $\Phi'$ such that $(aug_N(C), \Phi') \in$ NN$^*$ and $\Phi'|_C = \Phi$. There are two cases: either $\Phi'(l, final(C)) = \bot$ or not.

If $\Phi'(l, final(C)) = \bot$ then, by the definition of NN, $\Phi(l,u) = \bot$ for all $u \in V_C$ since $\bot \prec u \prec final(C)$. Thus, by Condition 2.3, $op_C(u) \neq W(l)$ for all $u \in V_C$. Thus, for any $T \in \mathcal{TS}(C)$, $\mathcal{W}_T(l,u) = \bot$ for all $u \in V_C$, as required.

Otherwise, let $w = \Phi'(l, final(C)) \in V_C$, $C'$ be the subcomputation of $C$ induced by $\{u \in V_C : \Phi(l,u) \neq w\}$, and $C''$ be the subcomputation of $C$ induced by $\{u \in V_C : \Phi(l,u) = w\}$. That is, $C'$ consists of nodes that do not observe $w$ and $C''$ consists of nodes that observe $w$.

Since $w \notin V_{C'}$, we have $|V_{C'}| < k$, so by the inductive hypothesis, a topological sort $T' \in \mathcal{TS}(C')$ exists such that $\Phi(l,u) = \mathcal{W}_{T'}(l,u)$ for all $u \in V_{C'}$. Let $T''$ be any topological sort of $C''$ that begins with $w$; such a topological sort exists because $v \not\prec w$ for all $v \in V_{C''}$ by Condition 2.2. Since $w$ is the only node of $C''$ that writes to $l$, $\mathcal{W}_{T''}(l,v) = w$ holds for all $v \in V_{C''}$. Let $T$ be the concatenation of $T'$ and $T''$. If we can prove that $T$ is a legitimate topological sort of $C$, then the claim is proven, since $\mathcal{W}_T = \Phi$ by construction of $T$.

To prove that $T \in \mathcal{TS}(C)$, we only need to show that $v \not\prec u$ for all $u \in V_{C'}$ and $v \in V_{C''}$. This property holds, because otherwise $v \prec u \prec \mathit{final}(C)$, and by the NN-dag consistency property, $\Phi'(l,u) = \Phi'(l,v) = w$ must hold since $\Phi'(l, \mathit{final}(C)) = \Phi'(l,v) = w$. But this conclusion contradicts the assumption that $u \in V_{C'}$. ∎

## 7 Discussion

This paper presents a computation-centric formal framework for defining and understanding memory models. The idea that the partial order induced by a program should be the basis for defining memory semantics, as opposed to the sequential order of instructions within one processor, already appears in the work by Gao and Sarkar on their version of location consistency [GS95]. Motivated by the experience with dag consistency [BFJ+96b, BFJ+96a, Joe96], we completely abstract away from a program, and assume the partial order (the "computation") as our starting point. *Post mortem* analysis has been used by [GK94] to verify (after the fact) that a given execution is sequentially consistent.

The need for formal frameworks for memory models has been felt by other researchers. Gibbons, Merrit, and Gharachorloo [GMG91] use the I/O automata model of Lynch and Tuttle [LT87] to give a formal specification of release consistency [GLL+90]. Later work [GM92] extends the framework to nonblocking memories. The main concern of these papers is to expose the architectural assumptions that are implicit in previous literature on relaxed memory models. In the present paper, rather than focusing on the correctness of specific implementations of a memory model, we are more interested in the formal properties of models, such as constructibility.

A different formal approach has been taken by the proponents of the $\lambda_S$ calculus [AMNS96], which is an extension of the $\lambda$ calculus with synchronization and side-effects. The $\lambda_S$ calculus gives a unified semantics of language *and* memory which is based on a set of rewriting rules. Preliminary $\lambda_S$ descriptions of sequential consistency [Lam79] and location consistency (in the sense of Definition 18) exist [Arv98].

Finally, many papers on memory models, starting with the seminal paper on sequential consistency [Lam79], have been written from an hardware viewpoint, without a strict formal framework. The reader is referred to [HP96] and [AG96] for good tutorials and further references on the subject. Gharachorloo [Gha95] also distinguishes *system-centric models*, which expose the programmer to the details of how a system may reorder operations, and *programmer-centric models*, which require the programmer to provide program-level information about the intended behavior of shared-memory operations but then allow the programmer to reason as if the memory were sequentially consistent. Both types of models, however, are processor-centric by our definition, since programs are still assumed to be sequential pieces of code running concurrently on several processors.

Historically, the abstract theory described in this paper arose from concrete problems in the context of research on dag consistency, a memory model for the Cilk multithreaded language for parallel computing [BJK+95, Blu95, Joe96]. Dag consistency was developed to capture formally the minimal guarantees that users of Cilk expected from the memory. It was formulated to forbid particular behaviors considered undesirable when programming in Cilk. This point of view can be thought of as looking for the weakest "reasonable" memory model. (See [Fri98] for a full discussion of this theme.) Dag consistency was also attractive because it is maintained by the BACKER algorithm used by Cilk, which has provably good performance [BFJ+96a].

Variants of dag consistency were developed to forbid "anomalies", or undesirable memory behaviors, as they were discovered. The papers [BFJ+96b] and [BFJ+96a] give two different definitions of dag consistency, which we call WW and WN. We were surprised to discover that WN is not constructible, and we tried both to find a "better" definition of dag consistency, and to capture the exact semantics of BACKER. Both problems have been solved. This paper presents a more or less complete picture of the various dag-consistent models and their mutual relationships. In another paper, Luchangco [Luc97] proves that BACKER supports location consistency. Consequently, the algorithmic analysis of [BFJ+96a] and the experimental results from [BFJ+96b] apply to location consistency with no change.

There are many possible directions in which this research can be extended. One obvious open problem is finding a simple characterization of NW$^*$ and WN$^*$. It would also be useful to investigate whether any algorithm can be found that is more efficient than BACKER that implements a weaker memory model than LC. Another direction is to formulate other consistency models in the computation-centric framework. Some models, such as release consistency [GLL+90], require computations to be augmented with locks, and how to do this is a matter of active research. Finally, as mentioned previously, it is important to develop an integrated theory of memory and language semantics.

## Acknowledgements

We are grateful to the many people who gave us suggestions and ideas for this paper. Charles Leiserson and Nancy Lynch gave us continual inspiration, support and encouragement. Bobby Blumofe, Chris Joerg, Charles Leiserson, and Keith Randall were part of the group that first tried to define and implement dag consistency. Bobby invented WW-dag consistency, initiating the whole topic. Observer functions are a formalization of Keith's idea to use a function as a device to force computation nodes to fix their viewpoint on memory. Finally, this paper benefits from many discussions with Arvind, Vivek Sarkar and Xiaowei Shen.

## References

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 1990.

[AMNS96] Arvind, J. W. Maessen, R. S. Nikhil, and Joe Stoy. Lambda-S: an implicitly parallel lambda-calculus with letrec, synchronization and side-effects. Technical report, MIT Laboratory for Computer Science, Nov 1996. Computation Structures Group Memo 393, also available at `http://www.csg.lcs.mit.edu:8001/pubs/csgmemo.html`.

[Arv98] Arvind. Personal communication, January 1998.

[BFJ$^+$96a] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[BFJ$^+$96b] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[BJK$^+$95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[Blu95] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Digest of Papers from the Thirty-Eighth IEEE Computer Society International Conference (Spring COMPCON)*, pages 528–537, San Francisco, California, February 1993.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[Fri98] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Massachusetts Institute of Technology, 1998.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, December 1995.

[GK94] P. B. Gibbons and E. Korach. On testing cache-coherent shared memories. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 177–188, Cape May, NJ, 1994.

[GLL$^+$90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, June 1990.

[GM92] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 306–315, 1992.

[GMG91] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, 1991.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface (SCI) Working Group, March 1989.

[GS95] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond memory coherence barrier. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 73–76, Oconomowoc, Wisconsin, August 1995.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[ISL96] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 277–287, Padua, Italy, June 1996.

[Joe96] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th Annual ACM Symposium on Principles of Distributed Computation*, pages 137–151, August 1987.

[Luc97] Victor Luchangco. Precedence-based memory models. In *Eleventh International Workshop on Distributed Algorithms*, number 1320 in Lecture Notes in Computer Science, pages 215–229. Springer-Verlag, 1997.

The following space intentionally left blank.