Precedence-Based Memory Models

Victor Luchangco*

M.I.T.

Abstract. This paper presents a general framework for understanding *precedence-based memory models*, which are generalizations of standard multiprocessor models. Precedence-based models need not mention processes explicitly, and can express any conditions that rely only on some operations being required to precede other operations. We define a generalized notion of *sequential consistency* and *per-location sequential consistency* in this framework, and we analyze the Backer algorithm used in the Cilk system [3], showing that it implements per-location sequential consistency. We also give conditions under which client processes cannot distinguish a per-location sequentially consistent memory from a sequentially consistent one.

1 Introduction

As distributed systems become ubiquitous, it becomes increasingly important to develop convenient ways to program these systems. Ideally, programs should express naturally the programmer's intention, and be easy to understand and reason about carefully. They should also be able to be implemented efficiently on multiprocessor systems, exploiting locality, re-ordering, and other techniques that mask disk and communication latency, to deliver high performance.

One common approach is to provide the processes with shared memory, which appears as though it is maintained by a single process. This provides programmers with relatively simple and intuitive semantics, called sequential consistency [11]. Unfortunately, maintaining such guarantees is expensive, even impossible on some systems. Thus, some systems are willing to tolerate some inconsistency in order to improve performance.

In order to reason about these systems, many weaker memory models have been proposed, such as processor consistency [9], release consistency [7], location consistency [6], scope consistency [10], eventual-serializability [5], dag-consistency [3], and others (see [1]). Unfortunately, these models are defined using different formalisms, and some of them do not even have formal definitions, making it hard to compare them rigorously.

Another drawback of sequential consistency and most of the models mentioned above is that the programmer must explicitly indicate which process issues each operation. Thus they cannot model systems such as Cilk [4, 13], which do not make processes directly accessible to the programmer.

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

This paper presents a general framework for understanding *precedence-based memory models*, which are generalizations of standard multiprocessor models. Precedence-based models allow clients to issue operations concurrently, specifying dependencies on other operations, but not exclusivity requirements. They need not mention processes explicitly, and can express any conditions that rely only on some operations being required to precede other operations. We believe this captures an interesting set of memory models that can be understood in a unified framework. This work is intended to provide structure to the field of modelling distributed memories, allowing us to categorize and compare models more easily, and also prove some general properties about them.

The memory may represent any deterministic serial data type with a generic set of operators, $\mathcal{O}$. However, to model distributed memories, operations are allowed to be concurrent, and may even be re-ordered by the system. The degree to which this is allowed defines the memory model. In the pure precedence-based memories described here, each operation can specify a set of operations that must precede it, thus defining a partial order of the requested operations.

In this framework, we define a generalized version of sequential consistency, and also *per-location sequential consistency*, which is similar to cache coherence in systems with caching. We demonstrate conditions under which the two are equivalent.

An important part of this work involves carefully specifying the serial semantics of the data, identifying characteristics that are important when concurrent operations are introduced. We consider how restrictions on the clients, or on the types of operations that can be applied to the data, can be used to guarantee greater consistency.

To demonstrate the utility of this framework, we formally specify and analyze the Backer algorithm used in the Cilk system [3], and we show that it implements per-location sequential consistency for read/write memories.

The rest of the paper is organized as follows: Section 2 defines some conventions used throughout the paper and Section 3 describes the formal model. Section 4 characterizes the serial semantics of the data. The general framework for precedence-based memories is presented in Section 5, and sequential consistency and per-location sequential consistency are defined in Section 6. Section 7 describes the Backer algorithm, and sketches the idea of its proof. Finally, Section 8 discusses related work and future directions for research.

2 Mathematical Conventions

We denote a sequence by $\langle a, b, c, \ldots \rangle$, and the empty sequence by ϵ. S^* denotes the set of finite sequences of a set S, and $S^+ = S^* - \{\epsilon\}$. The concatentation of sequences α and β is denoted by $\alpha \cdot \beta$. This notation is overloaded for adding a single element to a sequence, i.e., $e \cdot \alpha = \langle e \rangle \cdot \alpha$ and $\alpha \cdot e = \alpha \cdot \langle e \rangle$. We denote the ith element of α by α_i. The *restriction* $\alpha|_S$ of a sequence α to a set S is the subsequence of α consisting of all the elements of S in α. A sequence α is an *interleaving* of two sequences β and β' if β and β' are disjoint subsequences of α that together contain all the elements of α.

We denote the image of a set $S \subseteq A$ under $f : A \rightarrow B$ by $f(S) = \{f(a) : a \in S\}$, and the image of a sequence $\alpha \in A^*$ under f by $f(\alpha)$, i.e., $f(\alpha)_i = f(\alpha_i)$ for all i. We use 2^S to denote the power set of S. A partial function from A to B is denoted $f : A \rightarrow B_\perp$, where $B_\perp$ is the *lifted set* $B \cup \{\perp\}$. $\perp$ is not contained in non-lifted sets and indicates that the function is not defined at a value.

A *partial order* is any binary relation that is transitive and anti-symmetric; it need not be reflexive nor irreflexive. Two partial orders $\prec_1$ and $\prec_2$ are *consistent* if they do not order any two elements differently, i.e., for all distinct e, e', either $e \not\prec_1 e'$ or $e' \not\prec_2 e$. A partial order $\prec_1$ *includes* $\prec_2$ if $\prec_2 \subseteq \prec_1$. We use $\prec_1 \vee \prec_2$ to denote the transitive closure of the union of $\prec_1$ and $\prec_2$.

Theorem 1. *$\prec_1 \vee \prec_2$ is a partial order if and only if $\prec_1$ and $\prec_2$ are consistent.*

A *serialization* of a set S is a sequence that contains each element uniquely. A serialization α of S defines a total order $\prec_\alpha$ of S where $\alpha_i \prec_\alpha \alpha_j$ if $i \leq j$. It also partially orders any superset of S, where elements of S are ordered by $\prec_\alpha$ and all other elements are not ordered with respect to any elements. A serialization α is *consistent with* a partial order if $\prec_\alpha$ is, and it *includes* the partial order if $\prec_\alpha$ does.

3 Formal Model

We use a slight simplification of the I/O automaton of Lynch and Tuttle [12], ignoring the aspects related to liveness. An *non-live I/O automaton* A consists of:

- three disjoint sets of actions: in(A), out(A), and int(A);
- a set $states(A)$ of states;
- a nonempty subset $start(A)$ of start states;
- a set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of steps such that there exists $(s, \pi, s') \in steps(A)$ for all $s \in states(A)$, $\pi \in in(A)$.

We call the actions in $in(A)$, $out(A)$, and $int(A)$ the *input*, *output*, and *internal* actions respectively. The input and output actions are also called *external actions*, and the set of external actions is denoted by $ext(A)$. We denote the set of all actions of A by $acts(A) = in(A) \cup out(A) \cup int(A)$. We write $s \xrightarrow{\pi}_A s'$ or just $s \xrightarrow{\pi} s'$ as shorthand for $(s, \pi, s') \in steps(A)$.

An *execution fragment* $s_0, \pi_1, s_1, \pi_2, s_2, \ldots$ is a finite or infinite sequence of alternating states and actions such that $s_{i-1} \xrightarrow{\pi_i} s_i$ for all i. An *execution* is an execution fragment with $s_0 \in start(A)$. We denote the set of executions of A by $execs(A)$. A state is *reachable* in A if it appears in any execution of A. An *invariant* of A is a predicate that is true of every reachable state of A.

The *external image* of an execution fragment α is the subsequence $\alpha|_{ext(A)}$ of its external actions. A *trace* of A is the external image of an execution, and the set of traces is denoted by $traces(\alpha)$.

We often want to specify a distributed system by specifying the components that constitute the system. The entire system is then described by an automaton which is the *composition* of the automata describing the components. Informally, composition identifies actions with the same name at different component automata. Thus, when an action is executed, it is executed by all components with that action. The new automaton has the actions of all its components. There are some restrictions on the automata to be composed so that the composition makes sense. In particular, internal actions cannot be shared, and an action can be the output action of at most one component, and for technical reasons, actions cannot be shared by infinitely many components.

Formally, for any index set I, a set $\{A_i\}_{i \in I}$ of automata is *compatible* if $int(A_i) \cap acts(A_j) = \emptyset$ and $out(A_i) \cap out(A_j) = \emptyset$ for all $i, j \in I$ such that $i \neq j$, and no action is in $acts(A_i)$ for infinitely many $i \in I$. The *composition* $A = \Pi_{i \in I} A_i$ of a compatible set $\{A_i\}_{i \in I}$ of automata has the following components:

- $in(A) = \bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i)$
  $out(A) = \bigcup_{i \in I} out(A_i)$
  $int(A) = \bigcup_{i \in I} int(A_i)$
- $states(A) = \Pi_{i \in I} states(A_i)$
- $start(A) = \Pi_{i \in I} start(A_i)$
- $steps(A) = \{(s, \pi, s') : s_i \xrightarrow{\pi}_{A_i} s'_i \text{ or } \pi \notin acts(A_i) \wedge s_i = s'_i \text{ for all } i \in I\}$

We denote the composition of two compatible automata A and B by $A \circ B$.

For any $\alpha \in execs(\Pi_{i \in I} A_i)$, the *projection* $\alpha|_{A_i}$ onto A_i is the sequence α' consisting of alternating states and actions of A_i such that $\alpha'|_{acts(A_i)} = \alpha|_{acts(A_i)}$ and the states of α' are the ith component of the states in α preceding the actions in α'. Intuitively, the projection of α onto A_i is how α appears to A_i. For any $\beta \in traces(\Pi_{i \in I} A_i)$, its *projection* $\beta|_{A_i}$ onto A_i is the restriction $\beta|_{acts(A_i)}$ to the actions of A_i. We also write $execs(\Pi_{i \in I} A_i)|_{A_i}$ and $traces(\Pi_{i \in I} A_i)|_{A_i}$ for the sets of projections onto A_i of executions and traces of $\Pi_{i \in I} A_i$.

I/O automata can be used as specifications as well as implementations. We say that an automaton A *implements* another automaton B, and write $A \subseteq B$, if $in(A) = in(B)$, $out(A) = out(B)$, and $traces(A) \subseteq traces(B)$. We say that A and B are *equivalent*, and write $A \equiv B$, if they implement each other.

Theorem 2. *If $A_i \subseteq B_i$ for all $i \in I$ then $\Pi_{i \in I} A_i \subseteq \Pi_{i \in I} B_i$.*

A standard way to show that one automaton implements another is to use *simulations*, which establish a correspondence between the states of the two automata. Formally, if A and B are automata with $in(A) = in(B)$ and $out(A) = out(B)$ then a *simulation* from A to B is a relation f between $states(A)$ and $states(B)$ such that:

- If $s \in start(A)$ then there exists some $u \in start(B)$ such that $f(s, u)$.
- For reachable states s and u of A and B, if $f(s, u)$ and $s \xrightarrow{\pi}_A s'$, then there exists some u' such that $f(s', u')$ and there is some execution fragment of B from u to u' with the same external image as π.

Theorem 3. *If there is a simulation from A to B then $A \subseteq B$.*

4 Serial Data Type

This section introduces the formal framework to specify the serial semantics of shared objects. This is similar to the definitions of Lynch [12].

A *serial data type* $\mathcal{D}$ consists of a set Σ of states, an initial state $\hat{\sigma} \in \Sigma$, a set $\mathcal{O}$ of operators, a set $\mathcal{R}$ of possible return values, and two functions, $\tau_\Sigma : \Sigma \times \mathcal{O} \rightarrow \Sigma$ and $\tau_\mathcal{R} : \Sigma \times \mathcal{O} \rightarrow \mathcal{R}$, which define the state transitions and return values of each operator. As a shorthand, we write $\tau(\sigma, o) = (\tau_\Sigma(\sigma, o), \tau_\mathcal{R}(\sigma, o))$. We define the functions $\tau^*_\Sigma : \Sigma \times \mathcal{O}^* \rightarrow \Sigma$ and $\tau^+_\mathcal{R} : \Sigma \times \mathcal{O}^+ \rightarrow \mathcal{R}$ to yield the final state and return value of a sequence of operators applied in order. Formally, $\tau^*_\Sigma(\sigma, \epsilon) = \sigma$, $\tau^*_\Sigma(\sigma, \alpha \cdot o) = \tau_\Sigma(\tau^*_\Sigma(\sigma, \alpha), o)$, and $\tau^+_\mathcal{R}(\sigma, \alpha \cdot o) = \tau_\mathcal{R}(\tau^*_\Sigma(\sigma, \alpha), o)$.

Example 1. A read/write register with values V and initial value v_0 has $\Sigma = V$, $\hat{\sigma} = v_0$, $\mathcal{O} = \{read\} \cup \{write(v) : v \in V\}$, $\mathcal{R} = V \cup \{ack\}$, and τ such that $\tau(v, read) = (v, v)$ and $\tau(v, write(v')) = (v', ack)$.

We now make several definitions that are useful in the analysis later with concurrent accesses to the data. We say that an operator o is *oblivious* to an operator o' that does not affect its return value, i.e., $\tau^+_\mathcal{R}(\sigma, \langle o', o\rangle) = \tau_\mathcal{R}(\sigma, o)$ for all $\sigma \in \Sigma$. Two operators o and o' *commute* if the final state does not depend on the order in which they are applied, i.e., $\tau^*_\Sigma(\sigma, \langle o, o'\rangle) = \tau^*_\Sigma(\sigma, \langle o', o\rangle)$ for all $\sigma \in \Sigma$. Two sequences of operators commute if every operator of one commutes with every operator of the other. Two operators are *independent* if they commute and are oblivious to each other.

Example 2. For a read/write register, the *write* operators are oblivious to all operators, and the *read* operator commutes with all operators and is independent of itself.

The following lemma establishes some simple but useful results:

Lemma 4. *For all $o \in \mathcal{O}$ and $\alpha, \alpha' \in \mathcal{O}^*$:*

- *if o is oblivious to every operator in α then $\tau^+_\mathcal{R}(\sigma, \alpha \cdot o) = \tau_\mathcal{R}(\sigma, o)$.*
- *if o commutes with every operator in α then $\tau^*_\Sigma(\sigma, \alpha \cdot o) = \tau^*_\Sigma(\sigma, o \cdot \alpha)$.*
- *if α and α' commute then $\tau^*_\Sigma(\sigma, \beta) = \tau^*_\Sigma(\sigma, \alpha \cdot \alpha')$ for all $\sigma \in \Sigma$ and all interleavings β of α and α'.*

Some data objects may be viewed as a collection of "independent" objects, treated as a whole for convenience. Each component object may be considered to be at a different location. We formalize this intuition as follows:

Given a set $\mathcal{L}$ of locations, $f : \mathcal{O} \rightarrow \mathcal{L}$ is a *location partition* if operators mapped to different elements are independent, i.e., for all $o, o' \in \mathcal{O}$, if $f(o) \neq f(o')$ then o and o' are independent. Often, we use *loc* for location partitions, and write $o.loc$ instead of $loc(o)$. We say that $(\mathcal{D}, loc)$ is *location-based* if *loc* is a location partition. We say that o is performed at location $o.loc$. For $\alpha \in \mathcal{O}^*$ and $l \in \mathcal{L}$, we denote by $\alpha|_l$ the subsequence of α consisting of all operators performed at l. Similarly, if $S \subseteq \mathcal{O}$, we denote by $S|_l$ the subset of operators in S performed at l.

Lemma 5. *If $(\mathcal{D}, loc)$ is location-based then $\tau^+_{\mathcal{R}}(\sigma, \alpha \cdot o) = \tau^+_{\mathcal{R}}(\sigma, \alpha|_l \cdot o)$, where $l = o.loc$.*

Alternatively, the data type of such an object may be viewed as a composition of simpler data types. The full version of this paper includes a formal definition of composition for data types, but this is omitted here due to space limitations.

Example 3. A read/write memory with addresses A and values V is the composition of read/write registers with values V indexed by A. Specifically, $\Sigma = \Pi_{a \in A} V$, $\hat{\sigma} = (v_0)_{a \in A}$, $\mathcal{O} = \{read(a) : a \in A\} \cup \{write(a, v) : a \in A, v \in V\}$, $\mathcal{R} = V \cup \{ack\}$, and τ such that $\tau((v_{a'})_{a' \in A}, read(a)) = ((v_{a'})_{a' \in A}, v_a)$, and $\tau((v_{a'})_{a' \in A}, write(a, v)) = ((v'_{a'})_{a' \in A}, ack)$, where $v'_a = v$ and $v'_{a'} = v_{a'}$ for $a' \neq a$. An operator is said to be performed at its address.

Every *write* operator is oblivious to all operators, every *read* operator commutes with all operators and is independent of all *read* operators. Operators performed at different addresses are independent, so the function that maps each operator to its address is a location partition.

5 Precedence-Based Memories

This section lays out the basic framework for precedence-based memories, defining the interface between the memory which maintains the data object and the clients that wish to access it. We define the interface in a centralized fashion, with one automaton for the clients, and one for the memory. This allows us to formulate restrictions on the clients and memory as abstractly as possible, and to model systems which have nonlocal dependencies, or even systems with no explicit notion of processes. Although actual system implementations will typically have clients running on several processors, and a distributed implementation of the memory, these are merely particular implementations of these abstract automata. Processes and processors are not explicit in our abstract formulation.

5.1 Some Notation and Conventions

We assume that the memory is maintaining data of type $\mathcal{D}$. Memory operations consist of a request by the client to apply a data operator and a response by the memory system with the return value. To distinguish different requests of the same data operator, operations are tagged with identifiers from a set $\mathcal{I}$. No identifier may be used in more than one request. A request also specifies a set of identifiers of operations on which the requested operation depends, i.e., the operations that must precede it. To simplify notation, we denote an operation by its identifier, and assume there is a function $op : \mathcal{I} \to \mathcal{O}$ that maps each identifier to its associated operator, and that this function is statically determined.

If α is a sequence of unique identifiers containing id then $retval(id, \alpha) = \tau^+_{\mathcal{R}}(\hat{\sigma}, op(\alpha'))$, where α' is the prefix of α ending with id. This function gives the return value of an operation given the sequence of operations performed.

If $(\mathcal{D}, loc)$ is location-based then for $l \in \mathcal{L}$, $\alpha \in \mathcal{I}^*$, and $S \subseteq \mathcal{I}$, we use $S|_l = \{id \in S : op(id).loc = l\}$ and $\alpha|_l = \alpha|_{\mathcal{I}|_l}$. The following lemmas express some simple results for operations on location-based data:

Lemma 6. *For* $l \in \mathcal{L}$, $\alpha \in \mathcal{I}^*$, $op(\alpha)|_l = op(\alpha|_l)$

Lemma 7. *For* $\alpha \in \mathcal{I}^*$ *and* id *in* α, $retval(id, \alpha) = retval(id, \alpha|_{op(id).loc})$.

5.2 Clients

We introduce a generic automaton which expresses the well-formedness requirements on the clients accessing the memory. Informally, these requirements are that identifiers are unique and that operations depend only on operations that have already been requested. This prevents cyclic dependencies. We also maintain some useful bookkeeping variables.

Note that this one automaton models all the clients together. This allows us to specify more general and abstract programming systems which may not have any explicit notion of processes, such as the Cilk system [4, 13]. This is important because the specification of practical programming systems is an active area of research. Systems in which the process that issues each request is explicit can easily be modelled in this framework by incorporating the process identifier into the operation identifier, or even the operator.

Generic Clients Automaton: $\mathcal{GC}(\mathcal{O}, \mathcal{I})$

State

$Used$: a set of identifiers, initially empty.
$prev : \mathcal{I} \to 2^{\mathcal{I}}{}_{\perp}$; identifiers of client-specified preceding operations, initially all $\perp$.

Actions

Output $request(id, prev)$
Pre: $id \notin Used$
$prev \subseteq Used$
Eff: $Used \leftarrow Used \cup \{id\}$
$prev(id) \leftarrow prev$

Input $response(id, v)$
Eff: None

Since $\mathcal{O}$ and $\mathcal{I}$ are fixed, we usually drop them from the notation. A *clients automaton* is any automaton that implements $\mathcal{GC}$. We assume that every clients automaton has these state variables, and updates them in exactly this fashion.[1] It is easy to see that $Used$ is redundant since $Used = \{id : prev(id) \neq \perp\}$. We derive from $prev$ a partial order $\prec_c$ that is the reflexive and transitive closure of $\{(id, id') : id \in prev(id')\}$. We say that id and id' are *concurrent* in a state if they are not ordered by $\prec_c$.

[1] In fact, because they are only updated deterministically by external actions, the value of these variables in any reachable state can be determined from the trace leading to that state. These variables are useful for analysis, but a real implementation need not maintain them.

This generic automaton specifies a very large class of automata which can meaningfully interact with precedence-based memories. It is helpful to distinguish families of automata within this class about which we may be able to say more. In particular, it is useful to note clients that, when composed with a weak memory consistency model, behave as though they were composed with a sequentially consistent memory.

If $\mathcal{L}$ is a set of locations, then we say that a clients automaton C respects $f : \mathcal{I} \to \mathcal{L}$ if in every reachable state of C, for all concurrent operations id and id', we have $f(id) \neq f(id')$, i.e., C does not issue concurrent operations to the same location. If $f : \mathcal{O} \to \mathcal{L}$, we say that a client respects f if it respects $op \circ f$.

We are interested in the clients that respect location partitions. In particular, if $(\mathcal{D}, loc)$ is location-based then clients that respect loc will exhibit the same behaviors when composed with per-location sequentially consistent memory as when composed with globally sequentially consistent memory, as defined later in this paper.

5.3 Generic Precedence-Based Memory Automaton

We now present automaton for a generic precedence-based memory, without any restrictions on the return values of operations. This automaton maintains some data structures that are useful for understanding precedence-based memory, and provide notation and a framework with which to understand the various memory models. The only restrictions this automaton places on the behaviors arise from the order specified by the client.

Generic Precedence-Based Memory Automaton: $\mathcal{GPBM}(\mathcal{D}, \mathcal{I})$

State

$prev : \mathcal{I} \to 2^{\mathcal{I}}{}_{\perp}$; client-specified preceding operations, initially all $\perp$.
$pending \subseteq \mathcal{I}$; operations that still need a response, initially empty.
$done \subseteq \mathcal{I}$; operations that have been "done", initially empty.
$return\text{-}value : \mathcal{I} \to \mathcal{R}_{\perp}$; the return value for each operation, initially all $\perp$.

Actions

Input $request(id, prev)$
Eff: $pending \leftarrow pending \cup \{id\}$
$prev(id) \leftarrow prev$

Internal $do\text{-}operation(id, v)$
Pre: $id \in pending - done$
$prev(id) \subseteq done$
Eff: $done \leftarrow done \cup \{id\}$
$return\text{-}value(id) \leftarrow v$

Output $response(id, v)$
Pre: $id \in pending$
$v = return\text{-}value(id)$
Eff: $pending \leftarrow pending - \{id\}$

Since $\mathcal{D}$ and $\mathcal{I}$ are fixed, we usually drop them from the notation. A *memory automaton* is any automaton that implements $\mathcal{GPBM}$. We assume that every memory automaton has the *prev* and *pending* state variables, and updates them exactly as above.[2] We define $\prec_c$ and *concurrent* as we did for clients automata.[3]

Invariant 8. *For $\mathcal{GPBM}$:*

- $id \in pending \implies prev(id) \neq \bot$
- $id \in done \implies prev(id) \neq \bot \wedge prev(id) \subseteq done$
- $return\text{-}value(id) \neq \bot \implies id \in done$
- $id \in done \wedge id' \prec_c id \implies id' \in done$

5.4 The Generic Precedence-Based System

Lemma 9. *$\mathcal{GPBM}$ and $\mathcal{GC}$ are compatible.*

Invariant 10. *For $\mathcal{GPBM} \circ \mathcal{GC}$: $\mathcal{GPBM}.prev = \mathcal{GC}.prev$.*

Because of this invariant, we do not need to distinguish the *prev* variables, nor $\prec_c$ and *concurrent* which are derived from *prev*, of the memory and clients automata. This is true for any memory automaton M and clients automaton C.

6 Sequential Consistency

6.1 Global Sequential Consistency

In this section, we introduce a notion of sequential consistency generalized for arbitrary precedence-based memories. We specify this by an automaton, which rules out behaviors where operations predict what operations will be requested in the future. We present this automaton as an enhancement of the generic memory automaton. We include below only the *do-operation* action; the *request* and *response* actions are unchanged.

Lemma 11. *$g\mathcal{SC}(\mathcal{D}, \mathcal{I}) \subseteq \mathcal{GPBM}(\mathcal{D}, \mathcal{I})$*

Proof. The trivial relation which relates states with exactly the same state components is a simulation because *request* and *response* are identical in the two automata, and $do\text{-}operation(id, v, \alpha)$ in $g\mathcal{SC}$ simulates $do\text{-}operation(id, v)$ in $\mathcal{GPBM}$.

Invariant 12. *For $g\mathcal{SC}$: There is a serialization α of done consistent with $\prec_c$ such that $retval(id, \alpha) = return\text{-}value(id)$ for all $id \in done$.*

Proof. The serialization of the last *do-operation* action satisfies these conditions.

[2] As with clients automata, since these are updated deterministically by external actions, they are determined by the trace.

[3] Although *Used* is not a state variable of $\mathcal{GPBM}$, it can be derived from *prev* or (as noted) from the trace.

Global Sequential Consistency Automaton: $g\mathcal{SC}(\mathcal{D}, \mathcal{I})$

Actions (changes from $\mathcal{GPBM}$**)**

Internal *do-operation*(id, v, α)
Pre: $id \in pending - done$
$prev(id) \subseteq done$
α is a serialization of $done \cup \{id\}$ consistent with $\prec_c$
$\forall id' \in done,\ retval(id', \alpha) = return\text{-}value(id')$
$retval(id, \alpha) = v$
Eff: As before

6.2 Per-Location Sequential Consistency

For this section, we assume that $(\mathcal{D}, loc)$ is location-based. Intuitively, a per-location sequentially consistent memory maintains global sequential consistency among operations at the same location, but makes no guarantees for operations at different locations. This is similar to the coherence condition for cached systems. As before, we present this automaton as an enhancement of $\mathcal{GPBM}$, but notice the similarity to $g\mathcal{SC}$.

Per-Location Sequential Consistency Automaton: $pl\mathcal{SC}(\mathcal{D}, loc, \mathcal{I})$

Actions (changes from $\mathcal{GPBM}$**)**

Internal *do-operation*(id, v, α)
Pre: $id \in pending - done$
$prev(id) \subseteq done$
α is a serialization of $done|_{op(id).loc} \cup \{id\}$ consistent with $\prec_c$
$\forall id' \in done|_{op(id).loc},\ retval(id', \alpha) = return\text{-}value(id')$
$retval(id, \alpha) = v$
Eff: As before

Per-location sequential consistency can also be viewed, perhaps more naturally, as a composition of sequentially consistent memory locations. Intuitively, each operation gets "sent" to its location. To maintain the client-specified precedence, however, locations need to be informed of the existence and relative order in this precedence relation of operations at other locations. Thus, we can imagine that each operation gets "done" at its location, but a dummy operation with its identifier gets "sent" to all the other locations, so they can maintain the precedence relation.

6.3 Comparing $g\mathcal{SC}$ and $pl\mathcal{SC}$

Intuitively, we can see that $g\mathcal{SC}$ is stronger than $pl\mathcal{SC}$. It is also intuitive, but less obvious, that if the clients always explicitly order operations done at different locations, then they will not be able to distinguish the two types of memory. The following theorems formalize this intuition.

Theorem 13. $g\mathcal{SC}(\mathcal{D},\mathcal{I}) \subseteq pl\mathcal{SC}(\mathcal{D}, loc, \mathcal{I})$

Proof. The trival relation which relates states with the same state components is a simulation because the *request* and *response* actions are identical, and *do-operation*(id, v, α) in $g\mathcal{SC}$ simulates *do-operation*$(id, v, \alpha|_{op(id).loc})$ in $pl\mathcal{SC}$. This last condition follows since any partial order consistent with α is consistent with $\alpha|_l$, and $retval(id', \alpha|_l) = retval(id', \alpha)$ for all id', where $l = op(id').loc$.

Theorem 14. *If C respects loc then $pl\mathcal{SC}(\mathcal{D}, loc, \mathcal{I}) \circ C \subseteq g\mathcal{SC}(\mathcal{D},\mathcal{I}) \circ C$.*

Proof. (Sketch) Notice that because C respects *loc*, all operations on the same location are totally ordered by $\prec_c$. This means that for each location $l \in \mathcal{L}$, there is a unique serialization of the operations at l that is consistent with *prev*. Thus, this must be a subsequence of any serialization of *done* in $g\mathcal{SC}$. Since operators are oblivious to operators at different locations, the return values are determined by this subsequence, and these are the values that must be recorded in *return-value*.

7 Generic Backer Automaton

We now specify and analyze the Backer algorithm of [3]. The algorithm implements a per-location sequentially consistent read/write memory on a multiprocessor system with a cache for each process and a shared "backing store". Operations may depend explicitly on operations done at other processors. The coherence strategy is simple: An operation cannot be done unless all the operations it depends on are done at the same processor, or they have been committed to the backing store. In addition, a *read* or a *noop* must make sure the value is not in the cache if any operation it depends on is done by a different processor. That way, it will not be keeping a stale value. A processor may also flush the value back at any time, and may load the value whenever its own cache copy is not dirty. There are separate internal actions for the various types of operations, instead of a single *do-operation* action.

Because a read/write memory is a collection of read/write registers, it is sufficient to demonstrate that *Backer* implements $g\mathcal{SC}$ for a single register. This automaton models a set $\mathcal{P}$ of processors, each of which maintains a cache copy of a read/write register with values V, and, like $g\mathcal{SC}$ and $pl\mathcal{SC}$, is written as an enhancement of $\mathcal{GPBM}$. To analyze this automaton and show it is correct, we need to augment it with some auxiliary variables. We also combine the *read*, *write* and *noop* actions into a single *do-operation* action.

In order to prove that *Backer* implements $g\mathcal{SC}$, we need several invariants. We consider the major invariants and steps in the proof.

First, we notice that the *opseq*s are just serializations of the operations at each processor, or that have been committed.

Invariant 15. *For Backer:*

- $opseq(p)$ *is a serialization of* $uncommitted(p)$
- $opseq(\mathcal{B})$ *is a serialization of* $done - \bigcup_{p\in\mathcal{P}} uncommitted(p)$

Backer

Additional State Variables

$val : \mathcal{P} \to V_\perp$; initially all $\perp$	$proc : \mathcal{I} \to \mathcal{P}$; initially all $\perp$
$val(\mathcal{B}) :\to V$; initially v_0	$lastop : \mathcal{P} \to \mathcal{I}_\perp$; initially all $\perp$
$dirty : \mathcal{P} \to Bool$; initially all $true$	$opseq : \mathcal{P} \to \mathcal{I}^*$; initially all ϵ
$uncommitted : \mathcal{P} \to 2^{\mathcal{I}}$; initially all empty.	$opseq(\mathcal{B}) :\to \mathcal{I}^*$; initially ϵ

Actions

Internal $do\text{-}operation(id, p)$
Pre: $id \in pending - done$
$prev(id) \subseteq done$
$val(p) = \perp \wedge \forall id' \in prev(id),\ id' \notin uncommitted(proc(id'))$
or $\forall id' \in prev(id),\ proc(id') = p$
Eff: $proc(id) \leftarrow p$
if $op(id) = read$ then
if $val(p) = \perp$ then $val(p) \leftarrow val(\mathcal{B})$
$return\text{-}value(id) \leftarrow val(p)$
if $opseq(p) \neq \epsilon$ then append id to $opseq(p)$
if $lastop(p) = \perp$ then append id to $opseq(\mathcal{B})$
if $opseq(p) = \epsilon \wedge lastop(p) \neq \perp$ then
insert id after $lastop(p)$ in $opseq(\mathcal{B})$
$lastop(p) \leftarrow id$
if $op(id) = write(v)$ then
$val(p) \leftarrow v$
$dirty(p) \leftarrow true$
$return\text{-}value(id) \leftarrow ack$
append id to $opseq(p)$
$lastop(p) \leftarrow id$
if $op(id) = noop$ then
$return\text{-}value(id) \leftarrow ack$
if $opseq(p) \neq \epsilon$ then append id to $opseq(p)$
if $lastop(p) = \perp$ then append id to $opseq(\mathcal{B})$
if $opseq(p) = \epsilon \wedge lastop(p) \neq \perp$ then
insert id after $lastop(p)$ in $opseq(\mathcal{B})$
if $lastop(p) \neq \perp$ then $lastop(p) \leftarrow id$
if $dirty(p)$ then $uncommitted(p) \leftarrow uncommitted(p) \cup \{id\}$
$done \leftarrow done \cup \{id\}$

Internal $flush(p)$
Pre: None
Eff: if $dirty(p)$ then
$val(\mathcal{B}) \leftarrow val(p)$
$dirty(p) \leftarrow false$
$uncommitted(p) \leftarrow \emptyset$
append $opseq(p)$ to $opseq(\mathcal{B})$
$opseq(p) \leftarrow \epsilon$
$val(p) \leftarrow \perp$
$lastop(p) \leftarrow \perp$

Internal $load(p)$
Pre: $\neg dirty(p)$
Eff: $val(p) \leftarrow val(\mathcal{B})$
$lastop(p) \leftarrow$ the last element of $opseq(\mathcal{B})$

Proof. (Sketch) By induction on the length of an execution. Notice that $uncommitted(p)$ and $opseq(p)$ are modified together since $dirty(p) \iff opseq(p) \neq \epsilon$.

We now define $Opseq$ to be the set of sequences that are $opseq(\mathcal{B})$ followed by the concatenation of $opseq(p)$ for all $p \in \mathcal{P}$ in any order. (This is a derived state variable.) We will show that every serialization in $Opseq$ is a possible serialization for all the operations in $done$ that is consistent with the specified dependencies and the returned values.

Invariant 16. *For Backer: For all $\alpha \in Opseq$, α is a serialization of done.*

Define a partial order $\prec_{\mathcal{B}}$ such that $id \prec_{\mathcal{B}} id'$ if $id \prec_{\alpha} id'$ for all $\alpha \in Opseq$. We show that $\prec_{\mathcal{B}}$ includes $\prec_c$ for the operations in $done$.

Lemma 17. *For any reachable state s of Backer, if $s \xrightarrow{\pi} s'$ then $\prec_{\mathcal{B}}$ in s' includes $\prec_{\mathcal{B}}$ in s.*

Invariant 18. *For Backer: If $proc(id) = p$ and $lastop(p) \neq \bot$ then $id \prec_{\mathcal{B}} lastop(p)$.*

Invariant 19. *For Backer: For all $id, id' \in done$, if $id' \in prev(id)$ then $id' \prec_{\mathcal{B}} id$.*

Invariant 20. *For Backer: For all $\alpha \in Opseq$, α is consistent with $\prec_c$.*

Proof. Since $\alpha \in Opseq$, α is a serialization of $done$ that includes $\prec_{\mathcal{B}}$. Using the previous invariant, we can show that for all $id, id' \in done$, if $id \prec_c id'$ then $id \prec_{\mathcal{B}} id'$, and thus $id \prec_{\alpha} id'$, so $id' \not\prec_{\alpha} id$.

We now show that the serializations in $Opseq$ are consistent with the values returned. To do so, we define a function $assocval : \mathcal{I} \times \mathcal{I}^*$ as follows: If α is a sequence of unique identifiers containing id then $assocval(id, \alpha) = v$ if $op(id) = write(v)$ or $op(id) \in \{read, noop\}$ and either id is the first element of α and $v = v_0$ or $v = assocval(id', \alpha)$ where id' is the immediate predecessor of id in α. Another way of saying this is that $assocval(id, \alpha)$ is the value written immediately before id in α, or v_0 if there are no writes precede id in α.

First, we note that for any $id \in done$ and $\alpha \in Opseq$, $assocval$ depends only on the particular $opseq$ that contains id.

Invariant 21. *For Backer: For all $\alpha \in Opseq$, $id \in done$:*

- *If $id \in uncommitted(p)$ then $assocval(id, \alpha) = assocval(id, opseq(p))$*
- *If $id \in done - \bigcup_{p \in \mathcal{P}} uncommitted(p)$ then $assocval(id, \alpha) = assocval(id, opseq(\mathcal{B}))$*

Proof. (Sketch) This follows because the first element of $opseq(p)$ is always a $write$ if $opseq(p) \neq \epsilon$.

The next two invariants say that $assocval$ gives the value that would be read by a $read$ operation.

Invariant 22. *For Backer: If $opseq(p) \neq \epsilon$ then $lastop(p)$ is the last element of $opseq(p)$ and $val(p) = assocval(lastop(p), opseq(p))$.*

Invariant 23. *For Backer: If $opseq(\mathcal{B}) \neq \epsilon$ then $val(\mathcal{B}) = assocval(id, opseq(\mathcal{B}))$, where id is the last element of $opseq(\mathcal{B})$.*

Thus, the value returned by any *read* is the value of *assocval* for that operation for any $\alpha \in Opseq$:

Invariant 24. *For Backer: For all $id \in done$ such that $op(id) = read$, $return\text{-}value(id) = assocval(id, \alpha)$ for all $\alpha \in Opseq$.*

Theorem 25. *$Backer \subseteq g\mathcal{SC}$*

Proof. (Sketch) The trivial relation between states with identical *prev*, *pending*, *done*, and *return-value* state components is a simulation. To see this, notice that this is okay for the start states. The *request* and *response* actions simulate the same actions in $g\mathcal{SC}$, and the $do\text{-}operation(id, p)$ action simulates $do\text{-}operation(id, v, \alpha)$, where $\alpha \in Opseq$ and $v = retval(id, \alpha)$. The *flush* and *load* actions correspond to no action in $g\mathcal{SC}$. We know that the $do\text{-}operation(id, v, \alpha)$ action is enabled by the invariants. It is easy to check that the correspondence is maintained in the post-states of the two steps.

8 Discussion

We have presented a unified framework for understanding precedence-based memory models, which we hope can serve as a foundation to understanding more general memories. Because this framework is completely formal, we can prove the correctness of algorithms, and make rigorous comparisons between various proposed models. We believe that the careful definition and characterization of the serial semantics of data will also be helpful in understanding memory, and how algorithms can exploit specific classes of data, particular read/write memories.

Many people have proposed different memory models (see [1] for an overview), but only a few have proposed a unified framework that can be used to compare different models. Gibbons and Merritt [8] present a framework to specify non-blocking shared memories, and they do so at roughly the same level as we do. Attiya, et al [2] present a higher level framework which also considers the control operations in programs. However, both still model processes explicitly, and thus would not be able to model the Cilk system, for example. Furthermore, while it is important to be able to reason about programs eventually, rather than simply the sequence of operations actually requested of the memory, this requires some assumptions about the expressiveness of the programming language, and this is still an area of active research.

This work continues in the direction of Fekete, et al [5] and Blumofe, et al [3] in allowing memory models without explicit processes, but can still express models with explicit processes. It is not intended to express all memory models, however, since this task has proven to be very difficult. Rather, it is intended as a first step in trying to understand the essential properties of memory models in a coherent framework.

One direction that we believe will be very helpful to explore is how exclusivity requirements, such as mutual exclusion, or read/write locks, can be incorporated into this framework. A more modest goal would be to characterize different synchronization primitives in this framework, and whether the various primitives being proposed by be handled within this framework, or if they have some additional exclusivity requirements.

References

1. S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 9512, Rice University, Sept. 1995.
2. H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proc. of the Fifth ACM Symp. on Parallel Algorithms and Architectures*, June 1993.
3. R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proc. of the 10th Int'l Parallel Processing Symp.*, Honolulu, Hawaii, Apr. 1996.
4. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.
5. A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing*, pages 300–309, May 1996.
6. G. R. Gao and V. Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report 78, McGill University, ACAPS Laboratory, Dec. 1993.
7. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int'l Symp. on Computer Architecture*, pages 15–26, Seattle, Washington, June 1990.
8. P. Gibbons and M. Merritt. Specifying nonblocking shared memories. In *Proc. of the Fourth ACM Symp. on Parallel Algorithms and Architectures*, June 1992.
9. J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface (SCI) Working Group, Mar. 1989.
10. L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the Eighth ACM Symp. on Parallel Algorithms and Architectures*, June 1996.
11. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.
12. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Francisco, Calif., 1996.
13. Supercomputing Technologies Group. *Cilk 4.0 Reference Manual.* MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, June 1996.