December 3, 2001

SRC Research Report 176

Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking

Shaz Qadeer

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking

Shaz Qadeer

December 3, 2001

Abstract

The memory model of a shared-memory multiprocessor is a contract between the designer and programmer of the multiprocessor. The sequential consistency memory model specifies a total order among the memory (read and write) events performed at each processor. A trace of a memory system satisfies sequential consistency if there exists a total order of all memory events in the trace that is both consistent with the total order at each processor and has the property that every read event to a location returns the value of the last write to that location.

Descriptions of shared-memory systems are typically parameterized by the number of processors, the number of memory locations, and the number of data values. It has been shown that even for finite parameter values, verifying sequential consistency on general shared-memory systems is undecidable. We observe that, in practice, shared-memory systems satisfy the properties of causality and data independence. Causality is the property that values of read events flow from values of write events. Data independence is the property that all traces can be generated by renaming data values from traces where the written values are distinct from each other. If a causal and data independent system also has the property that the logical order of write events to each location is identical to their temporal order, then sequential consistency can be verified algorithmically. Specifically, we present a model checking algorithm to verify sequential consistency on such systems for a finite number of processors and memory locations and an arbitrary number of data values.

1 Introduction

Shared-memory multiprocessors are very complex computer systems. Multi-threaded programs running on shared-memory multiprocessors use an abstract view of the shared memory that is specified by a memory model. Examples of memory models for multiprocessors include sequential consistency [Lam79], partial store ordering [WG99], and the Alpha memory model [Com98]. The implementation of the memory model, achieved by a protocol running either in hardware or software, is one of the most complex aspects of multiprocessor design. These protocols are commonly referred to as cache-coherence protocols. Since parallel programs running on such systems rely on the memory model for their correctness, it is important to implement the protocols correctly. However, since efficiency is important for the commercial viability of these systems, the protocols are heavily optimized, making them prone to design errors. Formal verification of cache-coherence protocols can detect these errors effectively.

Descriptions of cache-coherence protocols are typically parameterized by the number of processors, the number of memory locations, and the number of data values. Verifying parameterized systems for arbitrary values of these parameters is undecidable for nontrivial systems. Interactive theorem proving is one approach to parameterized verification. This approach is not automated and is typically expensive in terms of the required human effort. Another approach is to model check a parameterized system for small values of the parameters. This is a good *debugging* technique that can find a number of errors prior to the more time-consuming effort of verification for arbitrary parameter values. In this paper, we present an automatic method based on model checking to verify that a cache-coherence protocol with fixed parameter values is correct with respect to the sequential consistency memory model.

The sequential consistency memory model [Lam79] specifies a *total order* among the memory events (reads and writes) performed locally at each processor. This total order at a processor is the order in which memory events occur at that processor. A trace of a memory system satisfies sequential consistency if there exists a total order of all memory events that is both consistent with the local total order at each processor, and has the property that every read to a location returns the latest (according to the total order) value written to that location. Surprisingly, verifying sequential consistency, even for fixed parameter values, is undecidable [AMP96]. Intuitively, this is because the witness total order could be quite different from the global temporal order of events for some systems. An event might need to be logically ordered after an event that occurs much later in a run. Hence any algorithm needs to keep track of a potentially unbounded history of a run.

In this paper, we consider the problem of verifying that a shared-memory system $S(n, m, v)$ with n processors, m locations and v data values is sequentially consistent. We present a method that can check sequential consistency for any fixed n and m and for arbitrary v. The correctness of our method depends on two assumptions —causality and data independence. The property of *causality* arises from the observation that protocols do not conjure up data values;

data is injected into the system by the initial values stored in the memory and by the writes performed by the processors. Therefore every read operation r to location l is associated with either the initial value of l or some write operation w to l that wrote the value read by r. The property of *data independence* arises from the observation that protocols do not examine data values; they just forward the data from one component of the system (cache or memory) to another. Since protocol behavior is not affected by the data values, we can restrict our attention, without loss of generality, to unambiguous runs in which the written data values to a location are distinct from each other and from the initial value. We have observed that these two assumptions are true of shared-memory systems that occur in practice [LLG+90, KOH+94, BDH+99, BGM+00].

For a causal and unambiguous run, we can deduce the association between a read and the associated write just by looking at their data values. This leads to a vast simplification in the task of specifying the witness total order for sequential consistency. It suffices to specify for each location, a total order on the writes to that location. By virtue of the association of write events and read events, the total order on the write events can be extended to a partial order on all memory events (both reads and writes) to that location. If a read event r reads the value written by the write event w, the partial order puts r *after* w and all write events preceding w, and *before* all write events succeeding w. As described before, sequential consistency specifies a total order on the memory events for each processor. Thus, there are n total orders, one for each processor, and m partial orders, one for each location, imposed on the graph of memory events of a run. A necessary and sufficient condition for the run to be sequentially consistent is that this graph is acyclic. We further show that existence of a cycle in this graph implies the existence of a *nice* cycle in which no two processor edges (imposed by the memory model) are for the same processor and no two location edges (imposed by the write order) are for the same location. This implies that a nice cycle can have at most $2 \times min(\{n, m\})$ edges; we call a nice cycle with $2 \times k$ edges a k-nice cycle. Further if the memory system is symmetric with respect to processor and location ids, then processor and location edges occur in a certain canonical order in the nice cycle. These two observations drastically reduce the number of cycles for any search.

We finally argue that a number of causal and data independent shared-memory systems occurring in practice also have the property that the witness write order at each location is simply the temporal order of the write events. In other words, a write event w is ordered before w' if w occurs before w'. We call this a simple write order, and it is in fact the correct witness for a number of shared-memory systems. For cache-based shared-memory systems, the intuitive explanation is that at any time there is at most one cache with write privilege to a location. The write privilege moves from one cache to another with time. Hence, the logical timestamps [Lam78] of the writes to a location order them exactly according to their global temporal order. We show that the proof that a simple write order is a correct witness for a memory system can be performed by model checking [CE81, QS81]. Specifically, the proof for the memory system $S(n, m, v)$ for fixed n and m and arbitrary v is broken into $min(\{n, m\})$ model checking lemmas, where the k-th lemma checks for the existence of canonical k-nice cycles.

The rest of the paper is organized as follows. Sections 2 and 3 formalize shared-memory systems and our assumptions of causality and data independence about them. Section 4 defines the sequential consistency memory model. Section 5 defines the notions of a witness and a constraint graph for an unambiguous and causal run. Section 6 and 7 show that it is sufficient to search for canonical nice cycles in the constraint graph. Section 8 shows how to use model checking to detect canonical nice cycles in the constraint graphs of the runs of a memory system. Finally, we discuss related work in Section 9 and conclude in Section 10.

2 Shared-memory systems

Let $\mathbb{N}$ denote the set of positive integers. For any $n \geq 1$, let $\mathbb{N}_n$ denote the set of positive integers up to n.

A memory system is parameterized by the number of processors, the number of memory locations, and the number of data values. In a memory system with n processors, m memory locations, and v data values, read and write events denoted by R and W can occur at any processor in $\mathbb{N}_n$, to any location in $\mathbb{N}_m$, and have any data value in $\mathbb{N}_v$. Formally, we define the following sets of events parameterized by the number of processors n, the number of locations m, and the number of data values v, where $n, m, v \geq 1$.

1. $E^r(n,m,v) = \{R\} \times \mathbb{N}_n \times \mathbb{N}_m \times \mathbb{N}_v$ is the set of *read events*.
2. $E^w(n,m,v) = \{W\} \times \mathbb{N}_n \times \mathbb{N}_m \times \mathbb{N}_v$ is the set of *write events*.
3. $E(n,m,v) = E^r(n,m,v) \cup E^w(n,m,v)$ is the set of *memory events*.
4. $E^a(n,m,v) \supseteq E(n,m,v)$ is the set of *all events*.
5. $E^a(n,m,v) \setminus E(n,m,v)$ is the set of *internal events*.

For every memory event $e = \langle a,b,c,d \rangle \in E(n,m,v)$, we define $op(e) = a$, $proc(e) = b$, $loc(e) = c$, and $data(e) = d$. The set of all finite sequences of events in $E^a(n,m,v)$ is denoted by $E^a(n,m,v)^*$. A *memory system* $S(n,m,v)$ is a regular subset of $E^a(n,m,v)^*$. A sequence $\sigma \in S(n,m,v)$ is a *run*.

Consider any $\sigma \in E^a(n,m,v)^*$. We denote the length of σ by $|\sigma|$ and the i-th element of σ by $\sigma(i)$. The set of indices of the memory events in σ is denoted by $dom(\sigma) = \{1 \leq k \leq |\sigma| \mid \sigma(k) \in E(n,m,v)\}$. The subsequence obtained by projecting σ onto $dom(\sigma)$ is denoted by $\overline{\sigma}$. If $\sigma \in S(n,m,v)$, the sequence $\overline{\sigma}$ is a *trace* of $S(n,m,v)$. A trace of $S(n,m,v)$ for any $n, m, v \geq 1$ is a trace of S. We define the following useful subsets of $dom(\sigma)$.

1. For all $1 \leq i \leq n$, the set of memory events by processor i denoted by $P(\sigma, i) = \{k \in dom(\sigma) \mid proc(\sigma(k)) = i\}$.

$typedef\ Msg\ \{m : \{ACKS, ACKX\}, a : \mathbb{N}_m, d : \mathbb{N}_v\} \cup \{m : \{INVAL\}, a : \mathbb{N}_m\};$
$typedef\ CacheEntry\ \{d : \mathbb{N}_v, s : \{INV, SHD, EXC\}\};$
$cache : array\ \mathbb{N}_n\ of\ array\ \mathbb{N}_m\ of\ CacheEntry;$
$inQ : array\ \mathbb{N}_n\ of\ Queue(Msg);$
$owner : array\ \mathbb{N}_m\ of\ \mathbb{N}_n \cup \{0\};$

Initial predicate
$\forall i \in \mathbb{N}_n, j \in \mathbb{N}_m : (cache[i][j].s = SHD \wedge inQ[i].isEmpty \wedge owner[j] \neq 0)$
Events

$\langle R, i, j, k\rangle$ $\quad cache[i][j].s \neq INV \wedge cache[i][j].d = k \rightarrow$
    "no op"
$\langle W, i, j, k\rangle$ $\quad cache[i][j].s = EXC \rightarrow$
    $cache[i][j].d := k$
$\langle ACKX, i, j\rangle$ $\quad cache[i][j].s \neq EXC \wedge owner[j] \neq 0 \rightarrow$
    $if\ owner[j] \neq i\ then\ cache[owner[j]][j].s := INV;$
    $owner[j] := 0;$
    for each $(p \in \mathbb{N}_n)$
        $if\ (p = i)\ then$
            $inQ[p] := append(inQ[p], \langle ACKX, j, cache[owner[j]][j].d\rangle)$
        $else\ if\ (p \neq owner[j] \wedge cache[p][j].s \neq INV)\ then$
            $inQ[p] := append(inQ[p], \langle INVAL, j\rangle)$
$\langle ACKS, i, j\rangle$ $\quad cache[i][j].s = INV \wedge owner[j] \neq 0 \rightarrow$
    $cache[owner[j]][j].s := SHD;$
    $owner[j] := 0;$
    $inQ[i] := append(inQ[i], \langle ACKS, j, cache[owner[j]][j].d\rangle);$
$\langle UPD, i\rangle$ $\quad \neg isEmpty(inQ[i]) \rightarrow$
    $let\ msg = head(inQ[i])\ in$
        $if\ (msg.m = INVAL)\ then$
            $cache[i][msg.a].s := INV$
        $else\ if\ (msg.m = ACKS)\ then\ \{$
            $cache[i][msg.a] := \langle SHD, msg.d\rangle;$
            $owner[msg.a] := i$
        $\}\ else\ \{$
            $cache[i][msg.a] := \langle EXC, msg.d\rangle;$
            $owner[msg.a] := i$
        $\}$
    $inQ[i] := tail(inQ[i])$

Figure 1: Example of memory system

2. For all $1 \leq i \leq m$, the set of memory events to location i denoted by $L(\sigma, i) = \{k \in dom(\sigma) \mid loc(\sigma(k)) = i\}$.

3. For all $1 \leq i \leq m$, the set of write events to location i denoted by $L^w(\sigma, i) = \{k \in L(\sigma, i) \mid op(\sigma(k)) = W\}$.

4. For all $1 \leq i \leq m$, the set of read events to location i denoted by $L^r(\sigma, i) = \{k \in L(\sigma, i) \mid op(\sigma(k)) = R\}$.

Example. Consider the memory system in Figure 1. It is a highly simplified model of the protocol used to maintain cache coherence within a single node in the Piranha chip multiprocessor system [BGM+00]. The system has three variables —*cache*, *inQ* and *owner*— and five events —the memory events $\{R, W\}$ and the internal events $\{ACKX, ACKS, UPD\}$. The variables *inQ* and *owner* need some explanation. For each processor i, there is an input queue $inQ[i]$ where incoming messages are put. The type of $inQ[i]$ is *Queue*. The operations *isEmpty*, *head* and *tail* are defined on *Queue*, and the operation *append* is defined on $Queue \times Msg$. They have the obvious meanings and their definitions have been omitted in the figure. For each memory location j, either $owner[j] = 0$ or $owner[j]$ contains the index of a processor. Each event is associated with a guarded command. The memory events R and W are parameterized by three parameters —processor i, location j and data value k. The internal events $ACKX$ and $ACKS$ are parameterized by two parameters —processor i and location j. The internal event UPD is parameterized by processor i. A *state* is a valuation to the variables. An *initial* state is a state that satisfies the initial predicate. An event is *enabled* in a state if the guard of its guarded command is true in the state. The variables are initialized to an initial state and updated by nondeterministically choosing an enabled event and executing the guarded command corresponding to it. A run of the system is any finite sequence of events that can be executed starting from some initial state.

A processor i can perform a read to location j if $cache[i][j].s \in \{SHD, EXC\}$, otherwise it requests $owner[j]$ for shared access to location j. The processor $owner[j]$ is the last one to have received shared or exclusive access to location j. The request by i has been abstracted away but the response of $owner[j]$ is modeled by the action $ACKS[i][j]$, which sends a $ACKS$ message containing the data in location j to i and temporarily sets $owner[j]$ to 0. Similarly, processor i can perform a write to location j if $cache[i][j].s = EXC$, otherwise it requests $owner[j]$ for exclusive access to location j. The processor $owner[j]$ responds by sending a $ACKX$ message to i and $INVAL$ messages to all other processors that have a valid copy of location j. $owner[j]$ is set to i when processor i reads the $ACKS$ or $ACKX$ message from $inQ[i]$ in the event $UPD[i]$. Note that new requests for j are blocked while $owner[j] = 0$. A processor i that receives an $INVAL$ message for location j sets $cache[i][j].s$ to INV. ■

3 Causality and data independence

In this section, we formalize our assumptions on memory systems. Each assumption is motivated by an observation about memory systems occurring in practice.

Memory systems do not conjure up data values; they move around data values that were introduced by initial values or write events. For example, in the memory system in Figure 1, only the write event W introduces a fresh data value in the system by updating the cache; the internal events $ACKS$, $ACKX$ and UPD move data around and the read event R reads the data present in the cache. Therefore, the data value of a read operation must either be the initial value or the value introduced by a write event. We can now formally state our first assumption.

Assumption 1 (Causality) *There is a function init mapping each trace of S to a function in $\mathbb{N} \to \mathbb{N}$ such that for all $n, m, v \geq 1$, traces τ of $S(n, m, v)$, and locations $1 \leq i \leq m$, if $x \in L^r(\tau, i)$, then either $data(\tau(x)) = init(\tau)(i)$ or there is $y \in L^w(\tau, i)$ such that $data(\tau(x)) = data(\tau(y))$.*

A function τ satisfying Assumption 1 is called an *initial function* of the parameterized memory system S. The initial function is used to model the initial values of the locations in the memory system. In the remainder of this paper, we fix a particular initial function *init* for S.

Memory systems also have the property that control decisions are oblivious to the data values. A cache line carries along with the actual program data a few state bits for recording whether it is in shared, exclusive or invalid mode. Typically, actions do not depend on the value of the data in the cache line. For example, in the memory system shown in Figure 1, there are no predicates involving the data fields of the cache lines and the messages in any of the internal events of the system. In such systems, renaming the data values of a run results in yet another run of the system. Moreover, every run can be obtained by data value renaming from a run in which the initial value and values of write events to any location i are all distinct from each other.

An unambiguous trace is one in which every write event to a location i has a value distinct from the initial value of i and the value of every other write to i. Thus, a read event can be paired with its source write event just by comparing data values. Formally, a trace τ of $S(n, m, v)$ is *unambiguous* if for all $x \in L^w(\tau, i)$, we have $data(\tau(x)) \neq init(\tau)(i)$ and $data(\tau(x)) \neq data(\tau(y))$ for all $y \in L^w(\tau, i) \setminus \{x\}$. The run σ is unambiguous if the trace $\overline{\sigma}$ is unambiguous.

For all $m, v, v' \geq 1$, a function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \to \mathbb{N}_v$ is called a *renaming function*. Intuitively, the function λ provides for each memory location c and data value d the renamed data value $\lambda(c, d)$. Let λ^d be a function on $E(n, m, v)$ such that for all $e = \langle a, b, c, d \rangle \in E(n, m)$, we have $\lambda^d(e) = \langle a, b, c, \lambda(c, d) \rangle$. The function λ^d is extended to sequences in $E(n, m, v)^*$ in the natural way. We can now state formally state our second assumption.

Assumption 2 (Data independence) *For all $n, m, v \geq 1$, we have that τ is a trace of $S(n, m, v)$ iff there is $v' \geq 1$, an unambiguous trace τ' of $S(n, m, v')$ and a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \rightarrow \mathbb{N}_v$ such that $\tau = \lambda^d(\tau')$ and $init(\tau)(j) = \lambda(j, init(\tau')(j))$ for all $1 \leq j \leq m$.*

Assumption 2 is motivated by the handling of data in typical cache-coherence protocols. This assumption can be syntactically enforced on protocol descriptions by imposing restrictions on the operations allowed on variables that contain data values [Nal99]. For example, one restriction can be that no data variable appears in the guard expression of an internal event or in the control expression of a conditional.

4 Sequential consistency

Suppose $S(n, m, v)$ is a memory system for some $n, m, v \geq 1$. The sequential consistency memory model [Lam79] is a correctness requirement on the traces of $S(n, m, v)$. In this section, we define sequential consistency formally.

We first define the simpler notion of a trace τ of $S(n, m, v)$ being serial. For all $1 \leq u \leq |\tau|$, let $lw(\tau, u)$ be the maximum element of the set $\{1 \leq k \leq u \mid op(\tau(k)) = W \wedge loc(\tau(k)) = loc(\tau(u))\}$ if the set is nonempty and 0 otherwise. In other words, the value of $lw(\tau, u)$ is the latest write event in τ to location $loc(\tau(u))$ occurring no later than u. If no such write event exists, then $lw(\tau, u)$ is 0. In particular, if u is a write event then $lw(\tau, u) = u$. The trace τ is *serial* if for all locations $1 \leq i \leq m$ and $u \in L(\tau, i)$,

$$\begin{array}{ll} data(\tau(u)) = init(\tau)(i), & \text{if } lw(\tau, u) = 0 \\ data(\tau(u)) = data(\tau(lw(\tau, u))), & \text{if } lw(\tau, u) \neq 0. \end{array}$$

Thus, a sequence is serial if every read to a location i returns the value of the latest write to i if one exists. Moreover, all reads to location i without a preceding write to i must return the initial value of i.

The *sequential consistency* memory model M is a function that maps every sequence of memory events $\tau \in E(n, m, v)^*$ and processor $1 \leq i \leq n$ to a total order $M(\tau, i)$ on $P(\tau, i)$ defined as follows: for all $u, v \in P(\tau, i)$, we have $\langle u, v \rangle \in M(\tau, i)$ iff $u < v$. A sequence τ is *sequentially consistent* if there is a permutation f on $\mathbb{N}_{|\tau|}$ such that the following conditions are satisfied.

C1 For all $1 \leq u, v \leq |\tau|$ and $1 \leq i \leq n$, if $\langle u, v \rangle \in M(\tau, i)$ then $f(u) < f(v)$.

C2 The sequence $\tau' = \tau_{f^{-1}(1)} \tau_{f^{-1}(2)} \ldots \tau_{f^{-1}(|\tau|)}$ is serial.

Intuitively, the sequence τ' is a permutation of the sequence τ such that the event at index u in τ is moved to index $f(u)$ in τ'. According to C1, this permutation must respect the total order $M(\tau, i)$ for all $1 \leq i \leq n$. According to C2, the permuted sequence must be serial. A run $\sigma \in S(n, m, v)$ is sequentially consistent if $\overline{\sigma}$ satisfies M. The memory system $S(n, m, v)$ is sequentially consistent iff every run of $S(n, m, v)$ is sequentially consistent.

The memory system in Figure 1 is sequentially consistent. Here is an example of a sequentially consistent run σ of that memory system, the corresponding trace τ of σ, and the sequence τ' obtained by permuting τ.

$$\sigma = \begin{array}{l} \langle ACKX, 1, 1\rangle \\ \langle UPD, 1\rangle \\ \langle W, 1, 1, 1\rangle \\ \langle R, 2, 1, 0\rangle \\ \langle UPD, 2\rangle \\ \langle ACKS, 2, 1\rangle \\ \langle UPD, 2\rangle \\ \langle R, 2, 1, 1\rangle \end{array} \qquad \tau = \overline{\sigma} = \begin{array}{l} \langle W, 1, 1, 1\rangle \\ \langle R, 2, 1, 0\rangle \\ \langle R, 2, 1, 1\rangle \end{array} \qquad \tau' = \begin{array}{l} \langle R, 2, 1, 0\rangle \\ \langle W, 1, 1, 1\rangle \\ \langle R, 2, 1, 1\rangle \end{array}$$

Sequential consistency orders the event $\tau(2)$ before the event $\tau(3)$ at processor 2. Let f be the permutation on $\mathbb{N}_3$ defined by $f(1) = 2$, $f(2) = 1$, and $f(3) = 3$. The sequence τ' is the permutation of τ under f. It is easy to check that both conditions C1 and C2 mentioned above are satisfied.

In order to prove that a run of a memory system is sequentially consistent, one needs to provide a reordering of the memory events of the run. This reordering should be serial and should respect the total orders imposed by sequential consistency at each processor. Since the memory systems we consider in this paper are data independent, we only need to show sequential consistency for the unambiguous runs of the memory system. This reduction is stated formally in the following theorem.

Theorem 4.1 *For all $n, m \geq 1$, the following statements are equivalent.*

1. *For all $v \geq 1$, every trace of $S(n, m, v)$ is sequentially consistent.*
2. *For all $v \geq 1$, every unambiguous trace of $S(n, m, v)$ is sequentially consistent.*

Proof: The (1) $\Rightarrow$ (2) case is trivial.

((2) $\Leftarrow$ (1)) Let τ be a trace of $S(n, m, v)$ for some $v \geq 1$. From Assumption 2 there is $v' \geq 1$, an unambiguous trace τ' of $S(n, m, v')$ and a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \rightarrow \mathbb{N}_v$ such that $\tau = \lambda^d(\tau')$ and $init(\tau)(j) = \lambda(j, init(\tau')(j))$ for all $1 \leq j \leq m$. Since τ' is sequentially consistent, we know that conditions C1 and C2 are satisfied by τ'. It is not difficult to see that both conditions C1 and C2 are satisfied by $\lambda^d(\tau')$ as well. Therefore τ is sequentially consistent. ∎

5 Witness

Theorem 4.1 states that in order to prove sequential consistency for all runs in a memory system with n processors and m locations (and any number of data values), it suffices to prove sequential consistency for all unambiguous runs in the system. In this section, we further reduce the problem of checking sequential consistency on an unambiguous run to the problem of detecting a cycle in a constraint graph.

Consider a memory system $S(n, m, v)$ for some fixed $n, m, v \geq 1$. A *witness* Ω for $S(n, m, v)$ maps every trace τ of $S(n, m, v)$ and location $1 \leq i \leq m$ to a total order $\Omega(\tau, i)$ on the set of writes $L^w(\tau, i)$ to location i. Then the total order $\Omega(\tau, i)$ on the write events to location i can be extended to a partial order $\Omega^e(\tau, i)$ on all memory events (including read events) to location i. If a read event r reads the value written by the write event w, the partial order puts r *after* w and all write events preceding w, and *before* all write events succeeding w. Formally, for every location $1 \leq i \leq m$, and $x, y \in L(\tau, i)$, we have that $\langle x, y\rangle \in \Omega^e(\tau, i)$ iff one of the following conditions holds.

1. $data(\tau(x)) = data(\tau(y))$, $op(\tau(x)) = W$, and $op(\tau(y)) = R$.
2. $data(\tau(x)) = init(\tau)(i)$ and $data(\tau(y)) \neq init(\tau)(i)$.
3. $\exists a, b \in L^w(\tau, i)$ such that $\langle a, b\rangle \in \Omega(\tau, i)$, $data(\tau(a)) = data(\tau(x))$, and $data(\tau(b)) = data(\tau(y))$.

We now show that the relation $\Omega^e(\tau, i)$ is a partial order. First, we need the following lemma about $\Omega^e(\tau, i)$.

Lemma 5.1 *For all unambiguous traces τ of $S(n, m, v)$, locations $1 \leq i \leq m$ and $r, s, t \in L(\tau, i)$, if $\langle r, s\rangle \in \Omega^e(\tau, i)$, then either $\langle r, t\rangle \in \Omega^e(\tau, i)$ or $\langle t, s\rangle \in \Omega^e(\tau, i)$.*

Proof: Since $\langle r, s\rangle \in \Omega^e(\tau, i)$, either $data(\tau(s)) \neq init(\tau)(i)$ or there is a $x \in L^w(\tau, i)$ such that $data(\tau(s)) = data(\tau(x))$. Since τ is an unambiguous trace, we have that $data(\tau(x)) \neq init(\tau)(i)$. Therefore, we get that $data(\tau(s)) \neq init(\tau)(i)$ in both cases. If $data(\tau(t)) = init(\tau)(i)$ we immediately get that $\langle t, s\rangle \in \Omega^e(\tau, i)$. So suppose $data(\tau(t)) \neq init(\tau)(i)$. Since τ is unambiguous, there is $y \in L^w(\tau, i)$ such that $data(\tau(t)) = data(\tau(y))$. We have three cases from the definition of $\langle r, s\rangle \in \Omega^e(\tau, i)$.

1. $data(\tau(r)) = data(\tau(s))$, $op(\tau(r)) = W$, and and $op(\tau(s)) = R$. Since Ω is a total order on $L^w(\tau, i)$, either $\langle r, y\rangle \in \Omega(\tau, i)$ or $\langle y, r\rangle \in \Omega(\tau, i)$. In the first case, we have $\langle r, t\rangle \in \Omega^e(\tau, i)$. In the second case, we have $\langle t, s\rangle \in \Omega^e(\tau, i)$.
2. $data(\tau(r)) = init(\tau)(i)$ and $data(\tau(s)) \neq init(\tau)(i)$. We get that $\langle r, t\rangle \in \Omega^e(\tau, i)$.
3. $\exists a, b \in L^w(\tau, i)$ such that $\langle a, b\rangle \in \Omega(\tau, i)$, $data(\tau(a)) = data(\tau(r))$, and $data(\tau(b)) = data(\tau(s))$. Since Ω is a total order on $L^w(\tau, i)$, either $\langle a, y\rangle \in \Omega(\tau, i)$ or $\langle y, a\rangle \in \Omega(\tau, i)$. In the first case, we have $\langle r, t\rangle \in \Omega^e(\tau, i)$. In the second case, we have by transitivity $\langle y, b\rangle \in \Omega(\tau, i)$ and therefore $\langle t, s\rangle \in \Omega^e(\tau, i)$.

■

Lemma 5.2 *For all unambiguous traces τ of $S(n, m, v)$ and locations $1 \leq i \leq m$, we have that $\Omega^e(\tau, i)$ is a partial order.*

Proof: We show that $\Omega^e(\tau, i)$ is irreflexive. In other words, for all $1 \le x \le |\tau|$, we have that $\langle x, x\rangle \notin \Omega^e(\tau, i)$. This is an easy proof by contradiction by assuming $\langle x, x\rangle \in \Omega^e(\tau, i)$ and performing a case analysis over the three resulting conditions.

We show that $\Omega^e(\tau, i)$ is anti-symmetric. In other words, for all $1 \le x < y \le |\tau|$, if $\langle x, y\rangle \in \Omega^e(\tau, i)$ then $\langle y, x\rangle \notin \Omega^e(\tau, i)$. We do a proof by contradiction. Suppose both $\langle x, y\rangle \in \Omega^e(\tau, i)$ and $\langle y, x\rangle \in \Omega^e(\tau, i)$. We reason as in the proof of Lemma 5.1 to obtain $data(\tau(x)) \ne init(\tau)(i)$ and $data(\tau(y)) \ne init(\tau)(i)$. Therefore there are $a, b \in L^w(\tau, i)$ such that $data(\tau(a)) = data(\tau(x))$ and $data(\tau(b)) = data(\tau(y))$. We perform the following case analysis.

1. $a = b$. Either $op(x) = R$ and $op(y) = R$, or $op(x) = W$ and $op(y) = R$, or $op(x) = R$ and $op(y) = W$. In the first case $\langle x, y\rangle \notin \Omega^e(\tau, i)$ and $\langle y, x\rangle \notin \Omega^e(\tau, i)$. In the second case $\langle y, x\rangle \notin \Omega^e(\tau, i)$. In the third case $\langle x, y\rangle \notin \Omega^e(\tau, i)$.

2. $\langle a, b\rangle \in \Omega(\tau, i)$. We have $data(\tau(x)) \ne data(\tau(y))$ since τ is unambiguous. Since $\Omega(\tau, i)$ is a total order, we have $\langle b, a\rangle \notin \Omega(\tau, i)$. Therefore $\langle y, x\rangle \notin \Omega^e(\tau, i)$.

3. $\langle b, a\rangle \in \Omega(\tau, i)$. This case is symmetric to Case 2.

Finally, we show that $\Omega^e(\tau, i)$ is transitive. Suppose $\langle x, y\rangle \in \Omega^e(\tau, i)$ and $\langle y, z\rangle \in \Omega^e(\tau, i)$. From Lemma 5.1, either $\langle x, z\rangle \in \Omega^e(\tau, i)$ or $\langle z, y\rangle \in \Omega^e(\tau, i)$. We have shown $\Omega^e(\tau, i)$ to be anti-symmetric. Therefore $\langle x, z\rangle \in \Omega^e(\tau, i)$. ∎

5.1 Constraint graph

Suppose τ is an unambiguous trace of $S(n, m, v)$. We have that $M(\tau, i)$ is a total order on $P(\tau, i)$ for all $1 \le i \le n$ from the definition of sequential consistency. For any witness Ω, we also have that $\Omega^e(\tau, j)$ is a partial order on $L(\tau, j)$ for all $1 \le j \le m$ from Lemma 5.2. The union of the n total orders $M(\tau, i)$ and m partial orders $\Omega^e(\tau, j)$ imposes a graph on $dom(\tau)$. The acyclicity of this graph, for some witness Ω, is a necessary and sufficient condition for the trace τ to satisfy sequential consistency. We define a function G that for every witness Ω returns a function $G(\Omega)$. The function $G(\Omega)$ maps every unambiguous trace τ of $S(n, m, v)$ to the graph $\langle dom(\tau), \bigcup_{1\le i\le n} M(\tau, i) \cup \bigcup_{1\le j\le m} \Omega^e(\tau, j)\rangle$. The work of Gibbons and Korach [GK97] defines a constraint graph on the memory events of a run that is similar to $G(\Omega)(\tau)$.

Theorem 5.3 *For all $n, m, v \ge 1$, every unambiguous trace of $S(n, m, v)$ is sequentially consistent iff there is a witness Ω such that the graph $G(\Omega)(\tau)$ is acyclic for every unambiguous trace τ of $S(n, m, v)$.*

Proof: ($\Rightarrow$) Suppose τ is an unambiguous trace of $S(n, m, v)$. Then τ satisfies sequential consistency. There is a permutation f on $\mathbb{N}_{|\tau|}$ such that conditions C1 and C2 are satisfied. For all $1 \le i \le m$, define $\Omega(\tau, i)$ to be the total order on $L^w(\tau, i)$ such that for all $x, y \in L^w(\tau, i)$, we have $\langle x, y\rangle \in \Omega(\tau, i)$ iff

$f(x) < f(y)$. We show that the permutation f is a linearization of the vertices in $G(\Omega)(\tau)$ that preserves all the edges. In other words, if $\langle x, y\rangle \in M(\tau, i)$ for some $1 \leq i \leq n$ or $\langle x, y\rangle \in \Omega^e(\tau, j)$ for some $1 \leq j \leq m$, then $f(x) < f(y)$. If $\langle x, y\rangle \in M(\tau, i)$ then we have from C1 that $f(x) < f(y)$. We show below that if $\langle x, y\rangle \in \Omega^e(\tau, j)$ then $f(x) < f(y)$.

Let $\tau' = \tau_{f^{-1}(1)}\tau_{f^{-1}(2)} \dots \tau_{f^{-1}(|\tau|)}$. For all $1 \leq u \leq |\tau|$ we have that $\tau(u) = \tau'(f(u))$. Since τ is unambiguous τ' is also unambiguous. Suppose $a \in L^w(\tau, j)$ and $x \in L(\tau, j)$. We show that if $data(\tau(a)) = data(\tau(x))$ then $f(a) \leq f(x)$. We have that $f(a) \in L^w(\tau', j)$, $f(x) \in L(\tau', j)$, $data(\tau(a)) = data(\tau'(f(a)))$, and $data(\tau(x)) = data(\tau'(f(x)))$. Since τ' is unambiguous, either $x = a$ or $op(\tau'(f(x))) = R$. In the first case $f(a) = f(x)$, and in the second case $f(a) = lw(\tau', f(x))$ which implies that $f(a) < f(x)$. Therefore $f(a) \leq f(x)$.

If $\langle x, y\rangle \in \Omega^e(\tau, j)$ then we have three cases. In each case, we show that $f(x) < f(y)$.

1. $data(\tau(x)) = data(\tau(y))$, $op(\tau(x)) = W$, and $op(\tau(y)) = R$. We have $data(\tau'(f(x))) = data(\tau'(f(y)))$, $op(\tau'(f(x))) = W$, and $op(\tau'(f(y))) = R$. Since τ' is unambiguous, we get that $data(\tau'(f(y))) \neq init(\tau')(j)$. Therefore $f(x) = lw(\tau', f(y))$ which implies that $f(x) < f(y)$.

2. $data(\tau(x)) = init(\tau)(j)$ and $data(\tau(y)) \neq init(\tau)(j)$. Since $x \neq y$ we have $f(x) \neq f(y)$. We show $f(x) < f(y)$ by contradiction. Suppose $f(y) < f(x)$. Since $data(\tau(y)) \neq init(\tau)(j)$ there is $b \in L^w(\tau, j)$ such that $data(\tau(b)) = data(\tau(y))$. Therefore we have that $f(b) \leq f(y) < f(x)$. Therefore $f(b) \leq lw(\tau', f(x))$. Since the trace τ' is unambiguous and $data(\tau'(f(x))) = init(\tau)(j)$ we have a contradiction.

3. $\exists a, b \in L^w(\tau, j)$ such that $\langle a, b\rangle \in \Omega(\tau, j)$, $data(\tau(a)) = data(\tau(x))$, and $data(\tau(b)) = data(\tau(y))$. We show $f(x) < f(y)$ by contradiction. Suppose $f(y) < f(x)$. We have that $f(a) \leq f(x)$ and $f(b) \leq f(y)$. Since $\langle a, b\rangle \in \Omega(\tau, j)$, we have $f(a) < f(b)$ from the definition of Ω. Thus we have $f(a) < f(b) \leq f(y) < f(x)$ Therefore $f(a) \neq lw(\tau', f(x))$. Since τ' is unambiguous and $data(\tau'(f(a))) = data(\tau'(f(x)))$ we have a contradiction.

($\Leftarrow$) Suppose there is a witness Ω such that $G(\Omega)(\tau)$ is acyclic for all unambiguous traces τ of $S(n, m, v)$. Let f be a linearization of the vertices in $G(\Omega)(\tau)$ that respects all edges. In other words, if $\langle x, y\rangle \in M(\tau, i)$ for some $1 \leq i \leq n$ or $\langle x, y\rangle \in \Omega^e(\tau, j)$ for some $1 \leq j \leq m$, then $f(x) < f(y)$. Then C1 is satisfied. Let τ' denote $\tau_{f^{-1}(1)}\tau_{f^{-1}(2)} \dots \tau_{f^{-1}(|\tau|)}$. We now show that τ' is serial.

We have that $\tau'(x) = \tau(f^{-1}(x))$ for all $1 \leq x \leq |\tau'|$. Let $loc(\tau'(x)) = j$ for some $1 \leq x \leq |\tau'|$. We show that if $lw(\tau', x) = 0$ then $data(\tau'(x)) = init(\tau)(j)$, and if $lw(\tau', x) \neq 0$ then $data(\tau'(x)) = data(\tau'(lw(\tau', x)))$. Thus, whenever $lw(\tau', x) = lw(\tau', y)$ we have $data(\tau'(x) = data(\tau'(y))$. Here are the two cases.

1. $lw(\tau', x) = 0$. We have that $op(\tau'(x)) = R$, otherwise $lw(\tau', x) = x \neq 0$ which is a contradiction. We prove by contradiction that $data(\tau'(x)) = init(\tau)(j)$. Suppose $data(\tau'(x)) \neq init(\tau)(j)$. Then there is $a \in L^w(\tau, j)$

such that $data(\tau'(x)) = data(\tau(f^{-1}(x))) = data(\tau(a))$. Therefore we get that $\langle a, f^{-1}(x)\rangle \in \Omega^e(\tau, j)$ which means that $f(a) < x$. This implies that $f(a) \leq lw(\tau', x)$ which is a contradiction.

2. $lw(\tau', x) \neq 0$. We show $data(\tau'(x)) = data(\tau'(lw(\tau', x)))$. If $op(\tau'(x)) = W$, then $lw(\tau', x) = x$ and $data(\tau'(x)) = data(\tau'(lw(\tau', x)))$. Suppose $op(\tau'(x)) = R$. Then there is $a \in L^w(\tau, j)$ such that $data(\tau'(x)) = data(\tau(f^{-1}(x))) = data(\tau(a))$. Therefore $\langle a, f^{-1}(x)\rangle \in \Omega^e(\tau, j)$ which means that $f(a) < x$. Therefore $f(a) \leq lw(\tau', x)$. Suppose $f(a) < lw(\tau', x)$. Then there is $b \in L^w(\tau, j)$ such that $f(a) < f(b) \leq x$. Since $f(a) < f(b)$, we have $\langle a, b\rangle \in \Omega^e(\tau, j)$. Therefore $\langle f^{-1}(x), b\rangle \in \Omega^e(\tau, j)$. This means that $x < f(b)$ which is a contradiction. Therefore $f(a) = lw(\tau', x)$.

■

5.2 Simple witness

Theorems 4.1 and 5.3 suggest that the memory system $S(n, m, v)$ can be proved sequentially consistent as follows. We produce for each $v' \geq 1$ a witness Ω for $S(n, m, v')$ and show for every unambiguous trace τ of $S(n, m, v')$ that the graph $G(\Omega)(\tau')$ is acyclic. But the construction of the witness is still left to the user. In this section, we argue that a simple witness, which orders the write events to a location exactly in the order in which they occur, suffices for a number of memory systems occurring in practice. Formally, a witness Ω is *simple* if for all traces τ of $S(n, m, v)$ and locations $1 \leq i \leq m$, we have $\langle x, y\rangle \in \Omega(\tau, i)$ iff $x < y$ for all $x, y \in L^w(\tau, i)$.

Consider the memory system of Figure 1. We argue informally that the simple witness is a good witness for this memory system. Permission to perform writes flows from one cache to another by means of the *ACKX* message. Note that for each location j, the variable *owner*$[j]$ is set to 0 (which is not the id of any processor) when an *ACKX* message is generated. When the *ACKX* message is received at the destination (by the *UPD* event), the destination moves to *EXC* state and sets *owner*$[j]$ to the destination id. A new *ACKX* message is generated only when *owner*$[j] \neq 0$. Thus, the memory system has the property that each memory location can be held in *EXC* state by at most one cache. Moreover, writes to the location j can happen only when the cache has the location in *EXC* state. Therefore, at most one cache can be performing writes to a memory location. This indicates that the logical order of the write events is the same as their temporal order. In other words, the simple witness is the correct witness for demonstrating that a run is sequentially consistent.

In general, for any memory system in which at any time at most one processor can perform write events to a location, the simple witness is very likely to be the correct witness. Most memory systems occurring in practice [LLG+90, KOH+94, BDH+99, BGM+00] have this property. In Section 8, we describe a model checking algorithm to verify the correctness of a memory system with respect to the simple witness. If the simple witness is indeed the desired witness and the memory system is designed correctly, then our algorithm will be able to verify its correctness. Otherwise, it will produce an error trace suggesting to the user that either there is an error in the memory system or the simple witness is not a correct witness. Thus our method for checking sequential consistency is clearly sound. We have argued that it is also complete on most shared-memory systems that occur in practice.

6 Nice cycle reduction

For some $n, m, v \geq 1$, let $S(n, m, v)$ be a memory system and Ω a witness for it. Let τ be an unambiguous trace of $S(n, m, v)$. In this section, we begin our quest for a method to detect cycles in $G(\Omega)(\tau)$. We show that it suffices to detect a special class of cycles called nice cycles. In Section 7, we further reduce our search for cycles to the class of canonical nice cycles. In Section 8, we will show that detection of canonical nice cycles can be performed by model checking.

We fix some $k \geq 1$ and use the symbol $\oplus$ to denote addition over the additive group with elements $\mathbb{N}_k$ and identity element k. A *k-nice* cycle in $G(\Omega)(\tau)$ is a sequence $u_1, v_1, \ldots, u_k, v_k$ of distinct vertices in $\mathbb{N}_{|\tau|}$ such that the following conditions are true.

1. For all $1 \leq x \leq k$, we have $\langle u_x, v_x \rangle \in M(\tau, i)$ for some $1 \leq i \leq n$ and $\langle v_x, u_{x \oplus 1} \rangle \in \Omega^e(\tau, j)$ for some $1 \leq j \leq m$.
2. For all $1 \leq x < y \leq k$ and for all $1 \leq i, j \leq n$, if $\langle u_x, v_x \rangle \in M(\tau, i)$ and $\langle u_y, v_y \rangle \in M(\tau, j)$ then $i \neq j$.
3. For all $1 \leq x < y \leq k$ and for all $1 \leq i, j \leq m$, if $\langle v_x, u_{x \oplus 1} \rangle \in \Omega^e(\tau, i)$ and $\langle v_y, u_{y \oplus 1} \rangle \in \Omega^e(\tau, j)$ then $i \neq j$.

In a k-nice cycle, no two edges belong to the relation $M(\tau, i)$ for any processor i. Similarly, no two edges belong to the relation $\Omega^e(\tau, j)$ for any location j. The above definition also implies that if a cycle is k-nice then $k \leq min(\{n, m\})$.

Theorem 6.1 *If the graph $G(\Omega)(\tau)$ has a cycle, then it has a k-nice cycle for some k such that $1 \leq k \leq min(\{n, m\})$.*

Proof: Suppose $G(\Omega)(\tau)$ has no k-nice cycles but does have a cycle. Consider the shortest such cycle $u_1, \ldots, u_l$ where $l \geq 1$. For this proof, we denote by $\oplus$ addition over the additive group with elements $\mathbb{N}_l$ and identity element l. Then for all $1 \leq x \leq l$ either $\langle u_x, u_{x \oplus 1} \rangle \in M(\tau, i)$ for some $1 \leq i \leq n$ or $\langle u_x, u_{x \oplus 1} \rangle \in \Omega^e(\tau, i)$ for some $1 \leq i \leq m$.

Since the cycle $u_1, \ldots, u_l$ is not k-nice for any k, there are $1 \leq a < b \leq l$ such that either (1) $\langle u_a, u_{a \oplus 1} \rangle \in M(\tau, i)$ and $\langle u_b, u_{b \oplus 1} \rangle \in M(\tau, i)$ for some $1 \leq i \leq n$, or (2) $\langle u_a, u_{a \oplus 1} \rangle \in \Omega^e(\tau, i)$ and $\langle u_b, u_{b \oplus 1} \rangle \in \Omega^e(\tau, i)$ for some $1 \leq i \leq m$.

Case (1). We have from the definition of M that $u_a < u_{a \oplus 1}$ and $u_b < u_{b \oplus 1}$. Either $u_a < u_b$ or $u_b < u_a$. If $u_a < u_b$ then $u_a < u_{b \oplus 1}$ or $\langle u_a, u_{b \oplus 1} \rangle \in M(\tau, i)$.

If $u_b < u_a$ then $u_b < u_{a\oplus 1}$ or $\langle u_b, u_{a\oplus 1}\rangle \in M(\tau, i)$. In both cases, we have a contradiction since the cycle can be made shorter.

Case (2). From Lemma 5.1, either $\langle u_a, u_b\rangle \in \Omega^e(\tau, i)$ or $\langle u_b, u_{a\oplus 1}\rangle \in \Omega^e(\tau, i)$. In both cases, we have a contradiction since the cycle can be made shorter. ■

7 Symmetry reduction

Suppose $S(n, m, v)$ is a memory system for some $n, m, v \geq 1$. In this section, we use symmetry arguments to further reduce the class of cycles that need to be detected in constraint graphs. Each k-nice cycle has $2 \times k$ edges with one edge each for k different processors and k different locations. These edges can potentially occur in any order yielding a set of isomorphic cycles. But if the memory system $S(n, m, v)$ is symmetric with respect to processor and memory location ids, presence of any one of the isomorphic nice cycles implies the existence of a nice cycle in which the edges are arranged in a canonical order. Thus, it suffices to search for a cycle with edges in a canonical order.

We discuss processor symmetry in Section 7.1 and location symmetry in Section 7.2. We combine processor and location symmetry to demonstrate the reduction from nice cycles to canonical nice cycles in Section 7.3.

7.1 Processor symmetry

For any permutation λ on $\mathbb{N}_n$, the function λ^p on $E(n, m, v)$ permutes the processor ids of events according to λ. Formally, for all $e = \langle a, b, c, d\rangle \in E(n, m, v)$, we define $\lambda^p(e) = \langle a, \lambda(b), c, d\rangle$. The function λ^p is extended to sequences in $E(n, m, v)^*$ in the natural way.

Assumption 3 (Processor symmetry) *For every permutation λ on $\mathbb{N}_n$ and for all traces τ of the memory system $S(n, m, v)$, we have that $\lambda^p(\tau)$ is a trace of $S(n, m, v)$ and $init(\lambda^p(\tau)) = init(\tau)$.*

We argue informally that the memory system in Figure 1 satisfies Assumption 3. The operations performed by the various parameterized actions on the state variables that store processor ids are symmetric. Suppose s is a state of the system. We denote by $\lambda^p(s)$ the state obtained by permuting the values of variables that store processors ids according to λ. Then, for example, if the action $UPD(i)$ in some state s yields state t, then the action $UPD(\lambda(i))$ in state $\lambda^p(s)$ yields the state $\lambda^p(t)$. Thus, from any run σ we can construct another run $\lambda^p(\sigma)$. If a shared-memory system is described with symmetric types, such as scalarsets [ID96], used to model variables containing processor ids, then it has the property of processor symmetry by construction.

The following lemma states that the sequential consistency memory model is symmetric with respect to processor ids. It states that two events in a trace τ ordered by sequential consistency remain ordered under any permutation of processor ids.

Lemma 7.1 *Suppose λ is a permutation on $\mathbb{N}_n$. Suppose τ and τ' are traces of $S(n,m,v)$ such that $\tau' = \lambda^p(\tau)$. Then for all $1 \leq x, y \leq |\tau|$, and for all $1 \leq i \leq n$, we have that $\langle x, y\rangle \in M(\tau, i)$ iff $\langle x, y\rangle \in M(\tau', \lambda(i))$.*

Proof: For all $1 \leq x, y \leq |\tau|$ and for all $1 \leq i \leq n$, we have that

$$\begin{array}{ll} & \langle x, y\rangle \in M(\tau, i) \\ \Leftrightarrow & \mathit{proc}(\tau(x)) = \mathit{proc}(\tau(y)) = i \text{ and } x < y \\ \Leftrightarrow & \mathit{proc}(\tau'(x)) = \mathit{proc}(\tau'(y)) = \lambda(i) \text{ and } x < y \\ \Leftrightarrow & \langle x, y\rangle \in M(\tau', \lambda(i)). \end{array}$$

■

The following lemma states that the partial order Ω^e obtained from a simple witness Ω is symmetric with respect to processor ids. It states that two events to location i ordered by $\Omega^e(\tau, i)$ in a trace τ remain ordered under any permutation of processor ids.

Lemma 7.2 *Suppose Ω is a simple witness for the memory system $S(n,m,v)$ and λ is a permutation on $\mathbb{N}_n$. Suppose τ and τ' are unambiguous traces of $S(n,m,v)$ such that $\tau' = \lambda^p(\tau)$. Then for all $1 \leq x, y \leq |\tau|$ and for all $1 \leq i \leq m$, we have that $\langle x, y\rangle \in \Omega^e(\tau, i)$ iff $\langle x, y\rangle \in \Omega^e(\tau', i)$.*

Proof: We have $\langle x, y\rangle \in \Omega(\tau, i)$ iff $x < y$ iff $\langle x, y\rangle \in \Omega(\tau', i)$. From the definition of $\Omega^e(\tau, i)$ we have the following three cases.

1. $\mathit{data}(\tau(x)) = \mathit{data}(\tau(y))$, $\mathit{op}(\tau(x)) = W$, $\mathit{op}(\tau(y)) = R$ iff $\mathit{data}(\tau'(x)) = \mathit{data}(\tau'(y))$, $\mathit{op}(\tau'(x)) = W$, $\mathit{op}(\tau'(y)) = R$.
2. $\mathit{data}(\tau(x)) = \mathit{init}(\tau)(i)$ and $\mathit{data}(\tau(y)) \neq \mathit{init}(\tau)(i)$ iff $\mathit{data}(\tau'(x)) = \mathit{init}(\tau')(i)$ and $\mathit{data}(\tau'(y)) \neq \mathit{init}(\tau')(i)$.
3. $\exists a, b \in L^w(\tau, i)$ such that $a < b$, $\mathit{data}(\tau(a)) = \mathit{data}(\tau(x))$, $\mathit{data}(\tau(b)) = \mathit{data}(\tau(y))$ iff $\exists a, b \in L^w(\tau', i)$ such that $a < b$, $\mathit{data}(\tau'(a)) = \mathit{data}(\tau'(x))$, $\mathit{data}(\tau'(b)) = \mathit{data}(\tau'(y))$.

■

7.2 Location symmetry

For any permutation λ on $\mathbb{N}_m$, the function λ^l on $E(n,m,v)$ permutes the location ids of events according to λ. Formally, for all $e = \langle a, b, c, d\rangle \in E(n,m,v)$, we define $\lambda^l(e) = \langle a, b, \lambda(c), d\rangle$. The function λ^l is extended to sequences in $E(n,m,v)^*$ in the natural way.

Assumption 4 (Location symmetry) *For every permutation λ on $\mathbb{N}_m$ and for all traces τ of the memory system $S(n,m,v)$, we have that $\lambda^l(\tau)$ is a trace of $S(n,m,v)$ and $\mathit{init}(\lambda^l(\tau)) \circ \lambda = \mathit{init}(\tau)$.*

We can argue informally that the memory system in Figure 1 satisfies Assumption 4 also. The operations performed by the various parameterized actions on the state variables that store location ids are symmetric. Suppose s is a state of the system. We denote by $\lambda^l(s)$ the state obtained by permuting the values of variables that store location ids according to λ. Then, for example, if the action $UPD(i)$ in some state s yields state t, then the action $UPD(\lambda(i))$ in state $\lambda^l(s)$ yields the state $\lambda^l(t)$. If scalarsets are used for modeling variables containing location ids, the shared-memory system will have the property of location symmetry by construction.

The following lemma states that the sequential consistency memory model is symmetric with respect to location ids. It states that two events in a trace τ ordered by sequential consistency remain ordered under any permutation of location ids.

Lemma 7.3 *Suppose λ is a permutation on $\mathbb{N}_m$. Suppose τ and τ' are traces of $S(n,m,v)$ such that $\tau' = \lambda^l(\tau)$. Then for all $1 \leq x, y \leq |\tau|$, and for all $1 \leq i \leq n$, we have that $\langle x, y\rangle \in M(\tau, i)$ iff $\langle x, y\rangle \in M(\tau', i)$.*

Proof: For all $1 \leq x, y \leq |\tau|$ and for all $1 \leq i \leq m$, we have that

$$\begin{array}{ll} & \langle x, y\rangle \in M(\tau, i) \\ \Leftrightarrow & proc(\tau(x)) = proc(\tau(y)) = i \text{ and } x < y \\ \Leftrightarrow & proc(\tau'(x)) = proc(\tau'(y)) = i \text{ and } x < y \\ \Leftrightarrow & \langle x, y\rangle \in M(\tau', i). \end{array}$$

■

The following lemma states that the partial order Ω^e obtained from a simple witness Ω is symmetric with respect to location ids. It states that two events to location i ordered by $\Omega^e(\tau, i)$ in a trace τ remain ordered under any permutation of location ids.

Lemma 7.4 *Suppose Ω is a simple witness for the memory system $S(n,m,v)$ and λ is a permutation on $\mathbb{N}_m$. Suppose τ and τ' are unambiguous traces of $S(n,m,v)$ such that $\tau' = \lambda^l(\tau)$. Then for all $1 \leq x, y \leq |\tau|$ and for all $1 \leq i \leq m$, we have that $\langle x, y\rangle \in \Omega^e(\tau, i)$ iff $\langle x, y\rangle \in \Omega^e(\tau', \lambda(i))$.*

Proof: We have $\langle x, y\rangle \in \Omega(\tau, i)$ iff $x < y$ iff $\langle x, y\rangle \in \Omega(\tau', \lambda(i))$. From the definition of $\Omega^e(\tau, i)$ we have the following three cases.

1. $data(\tau(x)) = data(\tau(y))$, $op(\tau(x)) = W$, $op(\tau(y)) = R$ iff $data(\tau'(x)) = data(\tau'(y))$, $op(\tau'(x)) = W$, $op(\tau'(y)) = R$.
2. $data(\tau(x)) = init(\tau)(i)$ and $data(\tau(y)) \neq init(\tau)(i)$ iff $data(\tau'(x)) = init(\tau')(\lambda(i))$ and $data(\tau'(y)) \neq init(\tau')(\lambda(i))$.
3. $\exists a, b \in L^w(\tau, i)$ where $a < b$, $data(\tau(a)) = data(\tau(x))$, and $data(\tau(b)) = data(\tau(y))$ iff $\exists a, b \in L^w(\tau', \lambda(i))$ where $a < b$, $data(\tau'(a)) = data(\tau'(x))$, and $data(\tau'(b)) = data(\tau'(y))$.

■

7.3 Combining processor and location symmetry

We fix some $k \geq 1$ and use the symbol $\oplus$ to denote addition over the additive group with elements $\mathbb{N}_k$ and identity element k. A k-nice cycle $u_1, v_1, \ldots, u_k, v_k$ is *canonical* if $\langle u_x, v_x \rangle \in M(\tau, x)$ and $\langle v_x, u_{x\oplus 1} \rangle \in \Omega^e(\tau, x \oplus 1)$ for all $1 \leq x \leq k$. In other words, the processor edges in a canonical nice cycle are arranged in increasing order of processor ids. Similarly, the location edges are arranged in increasing order of location ids. The following theorem claims that if the constraint graph of a run has a nice cycle then there is some run with a canonical nice cycle as well.

Theorem 7.5 *Suppose Ω is a simple witness for the memory system $S(n, m, v)$. Let τ be an unambiguous trace of $S(n, m, v)$. If the graph $G(\Omega)(\tau)$ has a k-nice cycle, then there is an unambiguous trace τ'' of $S(n, m, v)$ such that $G(\Omega)(\tau'')$ has a canonical k-nice cycle.*

Proof: Let $u_1, v_1, \ldots, u_k, v_k$ be a k-nice cycle in $G(\Omega)(\tau)$. Let $1 \leq i_1, \ldots, i_k \leq n$ and $1 \leq j_1, \ldots, j_k \leq m$ be such that $\langle u_x, v_x \rangle \in M(\tau, i_x)$ and $\langle v_x, u_{x\oplus 1} \rangle \in \Omega^e(\tau, j_{x\oplus 1})$ for all $1 \leq x \leq k$. Let α be a permutation on $\mathbb{N}_n$ that maps i_x to x for all $1 \leq x \leq k$. Then from Assumption 3 there is a trace τ' of $S(n, m, v)$ such that $\tau' = \alpha^p(\tau)$. Let β be a permutation on $\mathbb{N}_m$ that maps j_x to x for all $1 \leq x \leq k$. Then from Assumption 4 there is a trace τ'' of $S(n, m, v)$ such that $\tau'' = \beta^l(\tau')$. For all $1 \leq x \leq k$, we have that

$$
\begin{array}{lll}
 & \langle u_x, v_x \rangle \in M(\tau, i_x) & \\
\Leftrightarrow & \langle u_x, v_x \rangle \in M(\tau', \alpha(i_x)) = M(\tau', x) & \text{from Lemma 7.1} \\
\Leftrightarrow & \langle u_x, v_x \rangle \in M(\tau'', x) & \text{from Lemma 7.3.}
\end{array}
$$

For all $1 \leq x \leq k$, we also have that

$$
\begin{array}{lll}
 & \langle v_x, u_{x\oplus 1} \rangle \in \Omega^e(\tau, j_{x\oplus 1}) & \\
\Leftrightarrow & \langle v_x, u_{x\oplus 1} \rangle \in \Omega^e(\tau', j_{x\oplus 1}) & \text{from Lemma 7.2} \\
\Leftrightarrow & \langle v_x, u_{x\oplus 1} \rangle \in \Omega^e(\tau'', \beta(j_{x\oplus 1})) = \Omega^e(\tau'', x \oplus 1) & \text{from Lemma 7.4.}
\end{array}
$$

Therefore $u_1, v_1, \ldots, u_k, v_k$ is a canonical k-nice cycle in $G(\Omega)(\tau'')$. ∎

Finally, Theorems 4.1, 5.3, 6.1 and 7.5 can be easily combined to yield the following theorem.

Corollary 7.6 *Let $n, m \geq 1$. Suppose for all $v \geq 1$, for all unambiguous traces τ of $S(n, m, v)$, and for all $1 \leq k \leq min(\{n, m\})$, the graph $G(\Omega)(\tau)$ for the simple witness Ω does not have a canonical k-nice cycle. Then for all $v \geq 1$, every trace of $S(n, m, v)$ is sequentially consistent.*

8 Model checking memory systems

In this section, we present a model checking algorithm that, given a k such that $1 \leq k \leq min(\{n, m\})$, determines whether there is $v \geq 1$ and a trace τ in

Automaton $\mathit{Constrain}_k(j)$ for $1 \leq j \leq k$
States $\{a, b\}$
Initial state a
Accepting states $\{b\}$
Alphabet $E(n, m, 3)$
Transitions
$[]\ \neg(op(e) = W \wedge loc(e) = j)$
$\quad \rightarrow s' = s$
$[]\ s = a \wedge op(e) = W \wedge loc(e) = j \wedge data(e) = 1$
$\quad \rightarrow s' = a$
$[]\ s = a \wedge op(e) = W \wedge loc(e) = j \wedge data(e) = 2$
$\quad \rightarrow s' = b$
$[]\ s = b \wedge op(e) = W \wedge loc(e) = j \wedge data(e) = 3$
$\quad \rightarrow s' = b$

Automaton $\mathit{Constrain}_k(j)$ for $k < j \leq m$
States $\{a\}$
Initial state a
Accepting states $\{a\}$
Alphabet $E(n, m, 3)$
Transitions
$[]\ \neg(op(e) = W \wedge loc(e) = j) \vee data(e) = 1$
$\quad \rightarrow s' = s$

Automaton $\mathit{Check}_k(i)$ for $1 \leq i \leq k$
States $\{a, b, err\}$
Initial state a
Accepting states $\{err\}$
Alphabet $E(n, m, 3)$
Transitions
$[]\ s = a \wedge proc(e) = i \wedge loc(e) = i \wedge data(e) \in \{2, 3\}$
$\quad \rightarrow s' = b$
$[]\ s = b \wedge proc(e) = i \wedge loc(e) = i \oplus 1 \wedge (data(e) = 1 \vee (op(e) = W \wedge data(e) = 2))$
$\quad \rightarrow s' = err$
$[]\ \mathit{otherwise}$
$\quad \rightarrow s' = s$

Figure 2: Automata for detecting canonical k-nice cycle

$S(n, m, v)$ such that the graph $G(\Omega)(\tau)$ for the simple witness Ω has a canonical k-nice cycle. Corollary 7.6 then allows us to verify sequential consistency on the memory system $S(n, m, v)$ for all $v \geq 1$ by $min(\{n, m\})$ such model checking lemmas.

We fix some k such that $1 \leq k \leq min(\{n, m\})$. We use the symbol $\oplus$ to denote addition over the additive group with elements $\mathbb{N}_k$ and identity element k. The model checking algorithm makes use of m automata named $Constrain_k(j)$ for $1 \leq j \leq m$, and k automata named $Check_k(i)$ for $1 \leq i \leq k$. The automata are shown in Figure 2. Each automaton refers to a variable s that represents the state of the automaton. Model checking is performed on the system obtained by composing these automata with the memory system $S(n, m, 3)$.

We now define the regular languages accepted by these automata formally. In order to be concise, we use tuples of sets to denote the set obtained by taking the cross-product of the component sets. For example, the 4-tuple $\langle \{R, W\}, \{1\}, \{1\}, \{2, 3\} \rangle$ denotes the set $\{R, W\} \times \{1\} \times \{1\} \times \{2, 3\}$. This set denotes a read event or a write event by processor 1 to location 1 with data value 2 or 3. We further simplify notation and denote this set by $\langle \{R, W\}, 1, 1, \{2, 3\} \rangle$.

For all memory locations $1 \leq j \leq m$, the automaton $Constrain_k(j)$ constrains the write events to location j. If $1 \leq j \leq k$, then $Constrain_k(j)$ accepts sequences with a zero or more write events to location j with data value 1 followed by exactly one write event to location j with data value 2 followed by zero or more write events to location j with data value 3. Formally, the automaton $Constrain_k(j)$ accepts a sequence τ in $E(n, m, 3)^*$ iff the projection of τ to the alphabet $\langle W, \mathbb{N}_n, j, \mathbb{N}_3 \rangle$ satisfies the regular expression

$$\langle W, \mathbb{N}_n, j, 1 \rangle^* \cdot \langle W, \mathbb{N}_n, j, 2 \rangle \cdot \langle W, \mathbb{N}_n, j, 3 \rangle^*.$$

If $k < j \leq m$, then $Constrain_k(j)$ accepts sequences where all writes to location j have data value 1. Formally, the automaton $Constrain_k(j)$ accepts a sequence τ in $E(n, m, 3)^*$ iff the projection of τ to the alphabet $\langle W, \mathbb{N}_n, j, \mathbb{N}_3 \rangle$ satisfies the regular expression

$$\langle W, \mathbb{N}_n, j, 1 \rangle^*.$$

For all $1 \leq i \leq k$, there is an automaton $Check_k(i)$. The automaton $Check_k(i)$ accepts a trace τ if there are events x and y at processor i, with x occurring before y, such that x is an event to location i with data value 2 or 3 and y is an event to location $i \oplus 1$ with data value 1 or 2. Moreover, the event y is required to be a write event if its data value is 2. Formally,

$$\begin{array}{rcl} Check_k(i) & = & \begin{array}{l} (E(n, m, 3) \setminus \langle \{R, W\}, i, i, \{2, 3\} \rangle)^* \cdot \\ \langle \{R, W\}, i, i, \{2, 3\} \rangle \cdot \\ (E(n, m, 3) \setminus (\langle \{R, W\}, i, i \oplus 1, 1 \rangle \cup \langle W, i, i \oplus 1, 2 \rangle))^* \cdot \\ (\langle \{R, W\}, i, i \oplus 1, 1 \rangle \cup \langle W, i, i \oplus 1, 2 \rangle) \cdot \\ E(n, m, 3)^* \end{array} \end{array}$$

In order to check for canonical k-nice cycles, we compose the memory system $S(n, m, 3)$ with $Constrain_k(j)$ for all $1 \leq j \leq m$ and with $Check_k(i)$ for all $1 \leq i \leq k$. We use a model checker to determine if the resulting system has a trace in which the initial value of each memory location is 1.

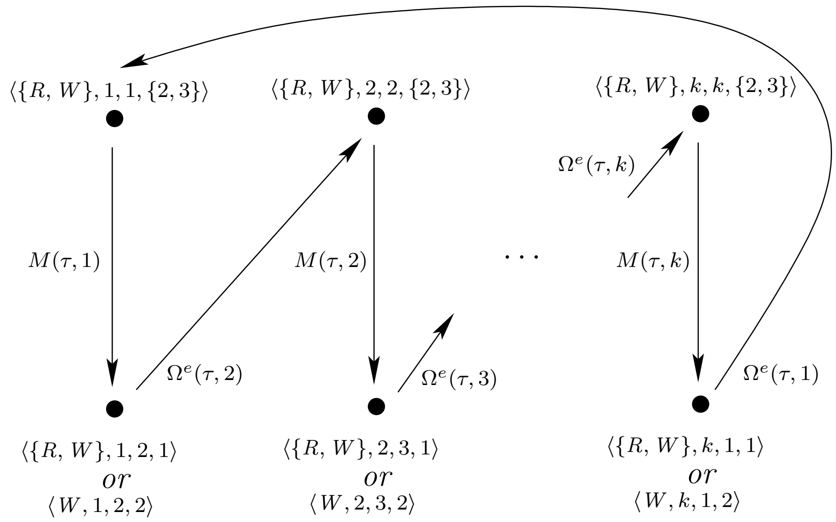

Figure 3: Canonical k-nice cycle

Any accepting run of the composed system has $2 \times k$ events which can be arranged as shown in Figure 3 to yield a canonical k-nice cycle. Each processor i for $1 \leq i \leq k$ and each location j for $1 \leq j \leq k$ supplies 2 events. Each event is marked by a 4-tuple denoting the possible values for that event. The edge labeled by $M(\tau, i)$ is due to the total order imposed by sequential consistency on the events at processor i. The edge labeled by $\Omega^e(\tau, j)$ is due to the partial order imposed by the simple witness on the events to location j. For example, consider the edge labeled $\Omega^e(\tau, 2)$ with the source event labeled by $\langle\{R, W\}, 1, 2, 1\rangle$ *or* $\langle W, 1, 2, 2\rangle$ and the sink event labeled by $\langle\{R, W\}, 2, 2, \{2, 3\}\rangle$. In any run of the composed system, the write events to location 2 with value 1 occur before the write event with value 2 which occurs before the write events with value 3. Since Ω is a simple witness, the partial order $\Omega^e(\tau, 2)$ orders all events labeled with 1 before all events labeled with 2 or 3. Hence any event denoted by $\langle\{R, W\}, 1, 2, 1\rangle$ is ordered before any event denoted by $\langle\{R, W\}, 2, 2, \{2, 3\}\rangle$. Moreover, the unique write event to location 2 with data value 2 is ordered before any other events with value 2 or 3. Hence the event $\langle W, 1, 2, 2\rangle$ is ordered before any event denoted by $\langle\{R, W\}, 2, 2, \{2, 3\}\rangle$.

We have given an intuitive argument above that a canonical k-nice cycle can be constructed from any run in the composed system. The following theorem proves that it is necessary and sufficient to check that the composed system has a run.

Theorem 8.1 *For all $n, m \geq 1$, there is $v \geq 1$ and a canonical k-nice cycle in $G(\Omega)(\tau)$ for the simple witness Ω and an unambiguous trace τ of $S(n, m, v)$ iff there is a trace τ' of $S(n, m, 3)$ such that $init(\tau')(j) = 1$ for all $1 \leq j \leq m$ and $\tau' \in Constrain_k(j)$ for all $1 \leq j \leq m$ and $\tau' \in Check_k(i)$ for all $1 \leq i \leq k$.*

Proof: ($\Rightarrow$) Suppose there is a canonical k-nice cycle $u_1, v_1, \ldots, u_k, v_k$ in the graph $G(\Omega)(\tau)$. Then $\langle u_x, v_x\rangle \in M(\tau, x)$ and $\langle v_x, u_{x\oplus 1}\rangle \in \Omega^e(\tau, x \oplus 1)$ for all $1 \leq x \leq k$. From the definition of $\Omega^e(\tau, x)$, we have that $\mathit{data}(\tau(u_x)) \neq \mathit{init}(\tau)(x)$ for all $1 \leq x \leq k$. Therefore, for all $1 \leq x \leq k$, there is a unique write event w_x such that $\mathit{data}(\tau(w_x)) = \mathit{data}(\tau(u_x))$.

For all $1 \leq j \leq m$, let V_j be the set of data values written by the write events to location j in τ, and let $f_j : V_j \to \mathbb{N}_{|\tau|}$ be the function such that $f_j(v)$ is the index of the unique write event to location j with data value v. We define a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_v \to \mathbb{N}_3$ as follows. For all $k < j \leq m$ and $x \in \mathbb{N}_v$, we have $\lambda(j, x) = 1$. For all $1 \leq j \leq k$ and $x \in \mathbb{N}_v$, we split the definition into two cases. For $x \in V_j$, we have

$$\lambda(j,x) = \begin{array}{ll} 1, & \text{if } f_j(x) < w_j \\ 2, & \text{if } f_j(x) = w_j \\ 3, & \text{if } f_j(x) > w_j. \end{array}$$

For $x \notin V_j$, we have

$$\lambda(j,x) = \begin{array}{ll} 1, & \text{if } x = \mathit{init}(\tau)(j) \\ 3, & \text{if } x \neq \mathit{init}(\tau)(j). \end{array}$$

From Assumption 2, there is a trace τ' of $S(n, m, 3)$ such that $\tau' = \lambda^d(\tau)$ and $\mathit{init}(\tau')(j) = \lambda(j, \mathit{init}(\tau)(j)) = 1$ for all $1 \leq j \leq m$. In τ', for every location j such that $1 \leq j \leq k$ every write event before w_j (including the initial value of j) has the data value 1, the write event at w_j has the data value 2, and the write events after w_j have the data value 3. Moreover, for every location j such that $k < j \leq m$ every write event has the data value 1. Therefore $\tau' \in \mathit{Constrain}_k(i)$ for all $1 \leq i \leq k$.

We show that $\tau' \in \mathit{Check}_k(i)$ for all $1 \leq i \leq k$. Since $\langle u_i, v_i\rangle \in M(\tau, i)$, we have that $u_i < v_i$ for all $1 \leq i \leq k$. We already have that $\mathit{data}(\tau'(u_i)) = \mathit{data}(\tau'(w_i)) = 2$ for all $1 \leq i \leq k$. Therefore all we need to show is that for all $1 \leq i \leq k$ we have $\mathit{data}(\tau'(v_i)) = 1$ or $\mathit{op}(\tau'(v_i)) = W$ and $\mathit{data}(\tau'(v_i)) = 2$. Since $\langle v_i, u_{i\oplus 1}\rangle \in \Omega^e(\tau, i \oplus 1)$, one of the following conditions hold.

1. $\mathit{data}(\tau(v_i)) = \mathit{data}(\tau(u_{i\oplus 1}))$, $\mathit{op}(\tau(v_i)) = W$, and $\mathit{op}(\tau(u_{i\oplus 1})) = R$. We have that $\mathit{op}(\tau'(v_i)) = \mathit{op}(\tau(v_i)) = W$. Since $\mathit{data}(\tau(v_i)) = \mathit{data}(\tau(u_{i\oplus 1}))$ we have $\mathit{data}(\tau'(v_i)) = \mathit{data}(\tau'(u_{i\oplus 1})) = 2$. Thus, we get $\mathit{op}(\tau'(v_i)) = W$ and $\mathit{data}(\tau'(v_i)) = 2$.

2. $\mathit{data}(\tau(v_i)) = \mathit{init}(\tau)(i \oplus 1)$ and $\mathit{data}(\tau(u_{i\oplus 1})) \neq \mathit{init}(\tau)(i \oplus 1)$. From the definition of λ, we get that $\mathit{data}(\tau'(v_i)) = 1$.

3. $\exists a \in L^w(\tau, i \oplus 1)$ such that $\langle a, w_{i\oplus 1}\rangle \in \Omega(\tau, i \oplus 1)$ and $\mathit{data}(\tau(a)) = \mathit{data}(\tau(v_i))$. Since $\langle a, b\rangle \in \Omega(\tau, i \oplus 1)$ and Ω is a simple witness we get $a < b$. Therefore $\lambda(i \oplus 1, \mathit{data}(\tau(a))) = 1$. Thus $\lambda(i \oplus 1, \mathit{data}(\tau(v_i))) = 1$ and $\mathit{data}(\tau'(v_i)) = 1$.

Thus, in all cases we have that either $\mathit{data}(\tau'(v_i)) = 1$ or $\mathit{op}(\tau'(v_i)) = W$ and $\mathit{data}(\tau'(v_i)) = 2$. Therefore $\tau' \in \mathit{Check}_k(i)$.

($\Leftarrow$) Suppose there is a trace τ' of $S(n, m, 3)$ such that $init(\tau')(j) = 1$ for all $1 \leq j \leq m$ and $\tau' \in Constrain_k(j)$ for all $1 \leq j \leq m$ and $\tau' \in Check_k(i)$ for all $1 \leq i \leq k$. For all $1 \leq i \leq k$, let $1 \leq u_i < v_i \leq |\tau'|$ be such that the automaton $Check_k(i)$ enters state b for the first time on observing $\tau'(u_i)$ and enters state err for the first time on observing $\tau'(v_i)$. Therefore we have $proc(\tau'(u_i)) = i$, $loc(\tau'(u_i)) = i$, and $data(\tau'(u_i)) \in \{2, 3\}$. We also have $proc(\tau'(v_i)) = i$, $loc(\tau'(v_i)) = i \oplus 1$, and either $data(\tau'(v_i)) = 1$ or $op(\tau'(v_i)) = W$ and $data(\tau'(v_i)) = 2$. From Assumption 2, there is a $v \geq 1$, an unambiguous trace τ of $S(n, m, v)$, and a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_v \to \mathbb{N}_3$ such that $\tau' = \lambda^d(\tau)$ and $\lambda(i, init(\tau)(i)) = init(\tau')(i) = 1$ for all $1 \leq i \leq m$. Therefore, we get that $data(\tau(u_i)) \neq init(\tau)(i)$ for all $1 \leq i \leq k$. We will show that $u_1, v_1, \ldots, u_k, v_k$ is a canonical k-nice cycle in $G(\Omega)(\tau)$. Since $proc(\tau(u_i)) = proc(\tau(v_i)) = i$ and $u_i < v_i$, we have $\langle u_i, v_i \rangle \in M(\tau, i)$ for all $1 \leq i \leq k$. We show that $\langle v_i, u_{i\oplus 1} \rangle \in \Omega^e(\tau, i \oplus 1)$ for all $1 \leq i \leq k$. First $loc(\tau(v_i)) = loc(\tau(u_{i\oplus 1})) = i \oplus 1$. For all $x, y \in L^w(\tau, i)$, if $\lambda(i, data(\tau(x)) < \lambda(i, data(\tau(y))$ then $x < y$ from the property of $Constrain_k(i)$. There are two cases on $\tau'(v_i)$.

1. $data(\tau'(v_i)) = 1$. We have that $\lambda(i \oplus 1, data(\tau(v_i))) = 1$. There are $a, b \in L^w(\tau, i \oplus 1)$ such that $data(\tau(a)) = data(\tau(v_i))$ and $data(\tau(b)) = data(\tau(u_{i\oplus 1}))$. Since $data(\tau'(a)) = 1$ and $data(\tau'(b)) \in \{2, 3\}$, we get from the definition of $Constrain_k(i \oplus 1)$ that $a < b$ or $\langle a, b \rangle \in \Omega(\tau, i \oplus 1)$. Therefore $\langle v_i, u_{i\oplus 1} \rangle \in \Omega^e(\tau, i \oplus 1)$.

2. $op(\tau'(v_i)) = W$ and $data(\tau'(v_i)) = 2$. We have that $op(\tau(v_i)) = W$. There is an event $b \in L^w(\tau, i \oplus 1)$ such that $data(\tau(b)) = data(\tau(u_{i\oplus 1}))$. There are two subcases: $data(\tau'(u_{i\oplus 1})) = 2$ or $data(\tau'(u_{i\oplus 1})) = 3$. In the first subcase, we have $v_i = b$ since $Constrain_k(i \oplus 1)$ accepts traces with a single write event labeled with 2. Therefore $data(\tau(v_i)) = data(\tau(u_{i\oplus 1}))$, $op(\tau(v_i)) = W$ and $op(\tau(u_{i\oplus 1})) = R$, and we get $\langle v_i, u_{i\oplus 1} \rangle \in \Omega^e(\tau, i \oplus 1)$. In the second subcase, since $data(\tau'(a)) = 2$ and $data(\tau'(b)) = 3$, we get from the definition of $Constrain_k(i \oplus 1)$ that $a < b$ or $\langle a, b \rangle \in \Omega(\tau, i \oplus 1)$. Therefore $\langle v_i, u_{i\oplus 1} \rangle \in \Omega^e(\tau, i \oplus 1)$.

Therefore $u_1, v_1, \ldots, u_k, v_k$ is a canonical k-nice cycle in $G(\Omega)(\tau)$. ∎

Example. We now give an example to illustrate the method described in this section. Although the memory system in Figure 1 is sequentially consistent, an earlier version had an error. The assignment $owner[j] := 0$ was missing in the guarded command of the action $\langle ACKS, i, j \rangle$. We modeled the system in TLA+ [Lam94] and model checked the system configuration with two processors and two locations using the model checker TLC [YML99]. The error manifests itself while checking for the existence of a canonical 2-nice cycle. First, the initial predicate of $S(2, 2, 3)$ is strengthened by conjoining it with the following predicate:

$$\forall i \in \mathbb{N}_n, j \in \mathbb{N}_m : (cache[i][j].d = 1).$$

This strengthening ensures that only those traces are examined where the initial value of every location is 1. Second, automata $Constrain_2(1)$, $Constrain_2(2)$,

$Check_2(1)$ and $Check_2(2)$ (from Figure 2) are composed with $S(2,2,3)$. Finally, the composed system is analyzed by the model checker TLC. The erroneous behavior is when the system starts in the initial state with all cache lines in *SHD* state and $owner[1] = owner[2] = 1$, and then executes the following sequence of 12 events:
1. $\langle ACKX, 2, 2\rangle$
2. $\langle UPD, 2\rangle$
3. $\langle ACKS, 1, 2\rangle$
4. $\langle ACKX, 2, 2\rangle$
5. $\langle ACKX, 1, 1\rangle$
6. $\langle UPD, 1\rangle$
7. $\langle UPD, 1\rangle$
8. $\langle W, 1, 1, 2\rangle$
9. $\langle R, 1, 2, 1\rangle$
10. $\langle UPD, 2\rangle$
11. $\langle W, 2, 2, 2\rangle$
12. $\langle R, 2, 1, 1\rangle$

After event 2, $owner[2] = 2$, $cache[1][2].s = INV$, and $cache[2][2].s = EXC$. Now processor 1 gets a shared ack message $\langle ACKS, 1, 2\rangle$ for location 2. Note that in the erroneous previous version of the example, this event does not set $owner[2]$ to 0. Consequently $owner[2] = 2$ and $cache[2][2].s = SHD$ after event 3. An exclusive ack to processor 2 for location 2 is therefore allowed to happen at event 4. Since the shared ack message to processor 1 in event 3 is still sitting in $inQ[1]$, $cache[1][2].s$ is still *INV*. Therefore event 4 does not generate an *INVAL* message to processor 1 for location 2. At event 5, processor 1 gets an exclusive ack message for location 1. This event also inserts an *INVAL* message on location 1 in $inQ[2]$ behind the *ACKX* message on location 2. After the *UPD* events to processor 1 in events 6 and 7, we have $cache[1][1].s = EXC$ and $cache[1][2].s = SHD$. Processor 1 writes 2 to location 1 and reads 1 from location 2 in the next two events, thereby sending automaton $Check_2(1)$ to the state *err*. Processor 2 now processes the *ACKX* message to location 2 in the *UPD* event 10. Note that processor 2 does not process the *INVAL* message to location 1 sitting in $inQ[2]$. At this point, we have $cache[2][1].s = SHD$ and $cache[2][2].s = EXC$. Processor 2 writes 2 to location 2 and reads 1 from location 1 in the next two events, thereby sending automaton $Check_2(2)$ to the state *err*. Since there has been only one write event of data value 2 to each location, the run is accepted by $Constrain_2(1)$ and $Constrain_2(2)$ also. ∎

Note that while checking for canonical k-nice cycles $Constrain_k(j)$ has 2 states for all $1 \leq j \leq k$ and 1 state for $k < j \leq m$. Also $Check_k(i)$ has 3 states for all $1 \leq i \leq k$. Therefore, by composing $Constrain_k(j)$ and $Check_k(i)$ with the memory system $S(n, m, 2)$ we increase the state of the system by a factor of at most $2^k \times 3^k$. Actually, for all locations $k < j \leq m$ we are restricting write events to have only the data value 1. Therefore, in practice we might reduce the set of reachable states.

9 Related work

Descriptions of shared-memory systems are parameterized by the number of processors, the number of memory locations, and the number of data values. The specification for such a system can be either an invariant or a shared-memory model. These specifications can be verified for some fixed values of the parameters or for arbitrary values of the parameters. The contribution of this paper is to provide a completely automatic method based on model checking to verify the sequential consistency memory model for fixed parameter values. We now describe the related work on verification of shared-memory systems along the two axes mentioned above.

A number of papers have looked at invariant verification. Model checking has been used for fixed parameter values [MS91, CGH$^+$93, EM95, ID96], while mechanical theorem proving [LD92, PD96] has been used for arbitrary parameter values. Methods combining automatic abstraction with model checking [PD95, Del00] have been used to verify snoopy cache-coherence protocols for arbitrary parameter values. McMillan [McM01] has used a combination of theorem proving and model checking to verify the directory-based FLASH cache-coherence protocol [KOH$^+$94] for arbitrary parameter values. A limitation of all these approaches is that they do not explicate the formal connection between the verified invariants and shared-memory model for the protocol.

There are some papers that have looked at verification of shared-memory models. Systematic manual proof methods [LLOR99, PSCH98] and theorem proving [Aro01] have been used to verify sequential consistency for arbitrary parameter values. These approaches require a significant amount of effort on the part of the user. Our method is completely automatic and is a good debugging technique which can be applied before using these methods. The approach of Henzinger *et al.* [HQR99] and Condon and Hu [CH01] requires a manually constructed finite state machine called the serializer. The serializer generates the witness total order for each run of the protocol. By model checking the system composed of the protocol and the serializer, it can be easily checked that the witness total order for every run is a trace of serial memory. This idea is a particular instance of the more general "convenient computations" approach of Katz and Peled [KP92]. In general, the manual construction of the serializer can be tedious and infeasible in the case when unbounded storage is required. Our work is an improvement since the witness total order is deduced automatically from the simple write order. Moreover, the amount of state we add to the cache-coherence protocol in order to perform the model checking is significantly less than that added by the serializer approach. The "test model checking" approach of Nalumasu *et al.* [NGMG98] can check a variety of memory models and is automatic. Their tests are sound but incomplete for sequential consistency. On the other hand, our method offers sound and complete verification for a large class of cache-coherence protocols.

Recently Glusman and Katz [GK01] have shown that, in general, interpreting sequential consistency over finite traces is not equivalent to interpreting it over infinite traces. They have proposed conditions on shared-memory systems under which the two are equivalent. Their work is orthogonal to ours and a combination of the two will allow verification of sequential consistency over infinite traces for finite parameter values.

10 Conclusions

We now put the results of this paper in perspective. Assumption 1 about causality and Assumption 2 about data independence are critical to our result that reduces the problem of verifying sequential consistency to model checking. Assumption 3 about processor symmetry and Assumption 4 about location symmetry are used to reduce the number of model checking lemmas to $min(\{n, m\})$ rather than exponential in n and m.

In this paper, the read and write events have been modeled as atomic events. In most real machines, each read or write event is broken into two separate events —a request from the processor to the cache, and a response from the cache to the processor. Any memory model including sequential consistency naturally specifies a partial order on the requests. If the memory system services processor requests in order then the order of requests is the same as the order of responses. In this case, the method described in this paper can be used by identifying the atomic read and write events with the responses. The case when the memory system services requests out of order is not handled by this paper.

The model checking algorithm described in the paper is sound and complete with respect to a simple witness for the memory system. In some protocols, for example the *lazy caching* protocol [ABM93], the correct witness is not simple. But the basic method described in the paper where data values of writes are constrained by automata can still be used if ordering decisions about writes can be made before the written values are read. The lazy caching protocol has this property and extending the methods described in the paper to handle it is part of our future work. We would also like to extend our work to handle other memory models.

Acknowledgments

Daithi O'Crualaoich pointed out the error in an earlier version of the protocol in Figure 1, thus providing a nice example to illustrate the method presented in this report. Yuan Yu helped with the use of TLC for model checking the protocol. Marcelo Glusman, Ranko Lazic, Sriram Rajamani and Yuan Yu provided extensive comments on earlier versions of this report.

References

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.

[AMP96] R. Alur, K.L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 219–228, 1996.

[Aro01] T. Arons. Using timestamping and history variables to verify sequential consistency. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 01: Computer-aided Verification*, Lecture Notes in Computer Science 2102, pages 423–435. Springer-Verlag, 2001.

[BDH+99] E. Bilir, R. Dickson, Y. Hu, M. Plakal, D. Sorin, M. Hill, and D. Wood. Multicast snooping: A new coherence method using a multicast address network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, 1999.

[BGM+00] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on sigle-chip multiprocessing. In *Proceedings of the 27st Annual International Symposium on Computer Architecture*, pages 282–293. IEEE Computer Society Press, 2000.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer-Verlag, 1981.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, pages 15–30, 1993.

[CH01] A.E. Condon and A.J. Hu. Automatable verification of sequential consistency. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 113–121, 2001.

[Com98] Alpha Architecture Committee. *Alpha Architecture Reference Manual.* Digital Press, 1998.

[Del00] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In E.A. Emerson and A.P. Sistla, editors, *CAV 2000: Computer Aided Verification*, Lecture Notes in Computer Science 1855, pages 53–68. Springer-Verlag, 2000.

[EM95] Á.Th. Eíriksson and K.L. McMillan. Using formal verification/analysis methods on the critical path in system design: a case study. In P. Wolper, editor, *CAV 95: Computer Aided Verification*, Lecture Notes in Computer Science 939, pages 367–380. Springer-Verlag, 1995.

[GK97] P.B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[GK01] M. Glusman and S. Katz. Extending memory consistency of finite prefixes to infinite computations. In K.G. Larsen and M. Nielsen, editors, *CONCUR 01: Theories of Concurrency*, Lecture Notes in Computer Science 2154, pages 411–425. Springer-Verlag, 2001.

[HQR99] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 301–315. Springer-Verlag, 1999.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313. IEEE Computer Society Press, 1994.

[KP92] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed program. *Communications of the ACM*, 21(7):558–565, 1978.

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

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LD92] P. Loewenstein and D.L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. *Formal Methods in System Design*, 1(4):355–383, 1992.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[LLOR99] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Distributed Computing*, 12(2/3):151–174, 1999.

[McM01] K.L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science 2144. Springer-Verlag, 2001.

[MS91] K.L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, pages 242–251, 1991.

[Nal99] R.P. Nalumasu. *Formal Design and Verification Methods for Shared Memory Systems*. PhD thesis, University of Utah, 1999.

[NGMG98] R.P. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 464–476. Springer-Verlag, 1998.

[PD95] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[PD96] S. Park and D.L. Dill. Protocol verification by aggregation of distributed transactions. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 300–310. Springer-Verlag, 1996.

[PSCH98] M. Plakal, D.J. Sorin, A.E. Condon, and M.D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, 1998.

[QS81] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.

[WG99] D.L. Weaver and T. Germond, editors. *The SPARC Architecture Manual*. Prentice Hall Inc., 1999.

[YML99] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *CHARME 99: IFIP Working Conference on Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science 1703, pages 54–66. Springer-Verlag, 1999.