Verifying Sequentially Consistent Memory
Problem Definition

Rob Gerth*
Eindhoven University of Technology†

April 1993

1 Introduction

In large multiprocessor architectures the design of efficient shared memory systems is important because the latency imposed on the processors when reading or writing should be kept at a minimum. This is usually achieved by interposing a *cache memory* between each processor and the shared memory system. A cache is private to a processor and contains a subset of the memory; hopefully containing most of the locations (variables) that the processor needs to access; i.e., the 'cache-hit' probability should be high. Such caches induce replication of data and hence there is a problem of *cache consistency*: if one processor updates the value at some location, all caches in the system that contain a copy of the location need to be updated. This is usually done by marking the location in the caches so that a subsequent access causes the location to be fetched from shared memory again, although variations exist. Clearly, changing a location and marking that location in other caches must be done as one atomic operation if memory is to behave as expected.

If the multiprocessor architecture is also distributed then such 'write and mark' operations cause unacceptable latencies. For instance, the DASH [LLG+92] and KSR1 [BFKR92] architectures envisage up to 10000 workstations to be connected and to operate on a conceptually shared memory. Atomic write-and-marks produce massive network congestion because at any time there will be many writes in progress.

The approach taken in such distributed shared memory architectures is to relax the constraints on the behavior of a standard shared memory. Most of these relaxations are patterned after Lamport's proposal of *sequential consistency* [Lam79]. In a standard memory the value that is read at a location must be the value that has last been written to

*Currently working in ESPRIT project 6021: "Building Correct Reactive Systems (REACT)"

†Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. Email: `robg@win.tue.nl`

that location. A sequentially correct memory satisfies a less stringent requirement: in Lamport's words

> *the result of any execution [of the memory] is the same as if the operations [memory accesses] of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

The challenge that sequentially correct memory poses is not so much the verification of yet another complex protocol but rather the fact that sequential consistency does not comfortably fit the patterns of standard refinement strategies (trace inclusion, failure or ready trace equivalence, testing pre order, bisimulation, etc.).

The aim of this exercise is to appraise how verifying sequential consistency can be accommodated for in a number of refinement methods. We do this by actually verifying a sequentially consistent memory—the lazy caching protocol of [ABM93]—using a variety of approaches. Although the protocol is proven correct in that paper, th proof is on a semantical level and is not grounded in a verification methodology. This makes the proof quite hard to follow and hard to generalize to more complex protocols such as release consistent or non-blocking memory.

2 Sequential consistency

In order to understand Lamport's definition, we first fix the behavior of a standard, 'serial' shared memory. This is done in Figures 1 and 2.

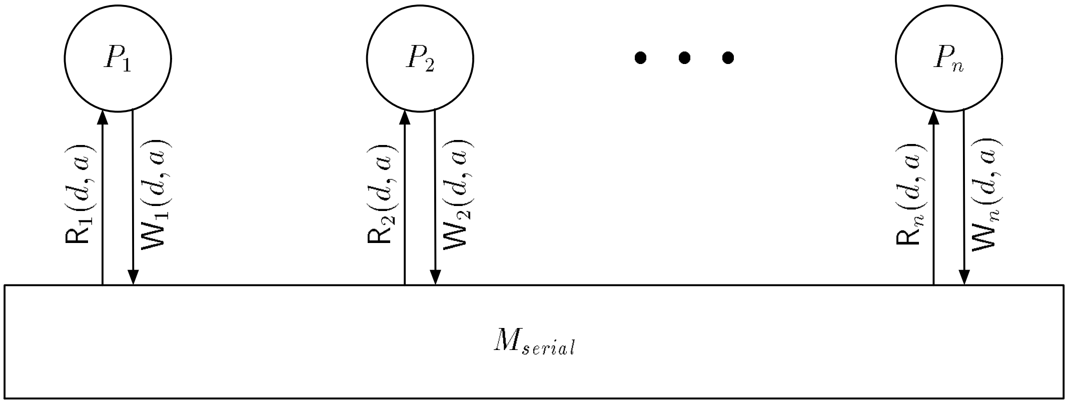

Figure 1: Architecture of M_{serial}

The interface of the memory comprises of read ($\mathsf{R}_i(d,a)$) and write ($\mathsf{W}_i(d,a)$) events for each processor P_i. The processors and the memory have to synchronize on these read and write events. The transition system in Figure 2 indicates that these are the only external events that M_{serial} participates in and that it has no internal events. A read event $\mathsf{R}_i(d,a)$, issued by P_i, can only occur if the memory holds value d at location a: $Mem[a] = d$. Write events $\mathsf{W}_i(d,a)$ can always occur with the expected result. The *external behavior* of the serial memory, Beh(M_{serial}), is defined as the maximal (hence infinite) sequences of read and write events generated according to the transition system of Figure 2. Hence, the memory *serializes* the reads and writes of the processors. The interface differs from the one used in [ABM93] as shall be discussed at the end of the introduction.

We can use this definition of serial memory both to characterize the sequential orders in which the memory accesses of the processors can be executed—any order that corresponds to a behavior of M_{serial}—as well as to characterize the order of operations of each individual processor—since a processor belongs to the environment of M_{serial}, possible orderings are determined by the behaviors of M_{serial} as well.

E	Event	Allowed if	Action
$\surd$	$\mathsf{R}_i(d,a)$	$Mem[a] = d$	
$\surd$	$\mathsf{W}_i(d,a)$		$Mem[a] := d$

initially: $\forall a \;\; Mem[a] = 0$

Figure 2: M_{serial}

We rephrase Lamport's proposal of correct behavior of sequentially consistent memory (SCM) thus

> *any external behavior, σ, [of the SCM] corresponds with an external behavior, τ, of M_{serial} so that the order in which the operations of each individual processor appear in σ coincides with order in which they appear in τ.*

For instance, the graph below depicts a possible prefix of a behavior of an SCM and a corresponding serial behavior:

SCM	$\mathsf{W}_1(1,x)$	$\mathsf{W}_2(2,y)$	$\mathsf{R}_3(2,y)$	$\mathsf{R}_3(0,x)$	$\mathsf{R}_3(1,x)$
P_1:	$\mathsf{W}_1(1,x)$				
P_2:		$\mathsf{W}_2(2,y)$			
P_3:			$\mathsf{R}_3(2,y)$	$\mathsf{R}_3(0,x)$	$\mathsf{R}_3(1,x)$
M_{serial}	$\mathsf{W}_2(2,y)$	$\mathsf{R}_3(2,y)$	$\mathsf{R}_3(0,x)$	$\mathsf{W}_1(1,x)$	$\mathsf{R}_3(1,x)$

Time flows from left to right. In particular notice that, although P_1 sets x to 1 before P_3 accesses that location, the first read of P_3 retrieves x's initial value 0. The effect of writes are thus seen to propagate slowly through the system. This is typical of sequentially consistent memory. Also notice that this SCM behavior is not possible for serial memory.

For completeness sake, we mention that the following behavior of the individual processes cannot be accommodated for by SCM:

$$\begin{array}{llllll}
P_1: & \mathsf{W}_1(1,x) & & & & \\
P_2: & & \mathsf{W}_2(2,x) & & & \\
P_3: & & & \mathsf{R}_3(1,x) & & \mathsf{R}_3(2,x) \\
P_4: & & & & \mathsf{R}_4(2,x) & & \mathsf{R}_4(1,x)
\end{array}$$

The problem is that P_3 and P_4 'observe' the writes of P_1 and P_2 in different order.

Sequential consistency has been the canonical distributed memory model for a long time. In practice, however, different, still weaker memory models tend to be implemented as the synchronization overhead of SCM is still too large. For instance, the *processor consistency* model would allow the above behavior at the processors. See [Mos93] for an overview of distributed memory models.

A formal definition

Let $\cdot\lceil i$ denote the operation on behaviors of removing the events that do not originate from process P_i or that are not external. Then we have

A memory M is sequentially consistent w.r.t. M_{serial}, M *s.c.* M_{serial}, iff

$$\forall \sigma \in \mathrm{Beh}(\mathrm{M})\ \exists \tau \in \mathrm{Beh}(M_{serial})\ \forall i = 1 \ldots n \quad \sigma\lceil i = \tau\lceil i$$

`Computation model`

3 The lazy caching protocol

In [ABM93] a sequential correct memory that is not serial was proposed: the lazy caching protocol. We use a slightly adapted version of this protocol.

The architecture of M_{distr} is depicted in Figure 3; the transition system in Figure 4. The protocol is thus geared towards a bus based architecture. Here, too, the interface of the memory comprises of the read and write events of the processors. M_{distr}, however, interposes caches C_i between the shared memory Mem and the processes P_i. Each cache C_i contains a part of the memory Mem and has two queues associated with it: an out-queue Out_i in which P_i's write requests are buffered and an in-queue In_i in which the pending cache updates are stored. These queues model the asynchronous behavior of write events in a sequential consistent memory. The gray arrows indicate the information flows from the out queues to the in queues and to Mem.

A write event $\mathsf{W}_i(d,a)$ does not have immediate effect. Instead, a request (d,a) is placed in Out_i. When the write request is taken out of the queue, by an internal memory-write event $\mathsf{MW}_i(d,a)$, the memory is updated and a cache update request (d,a) is placed in every in-queue. This cache update is eventually removed from the top of some queue In_j

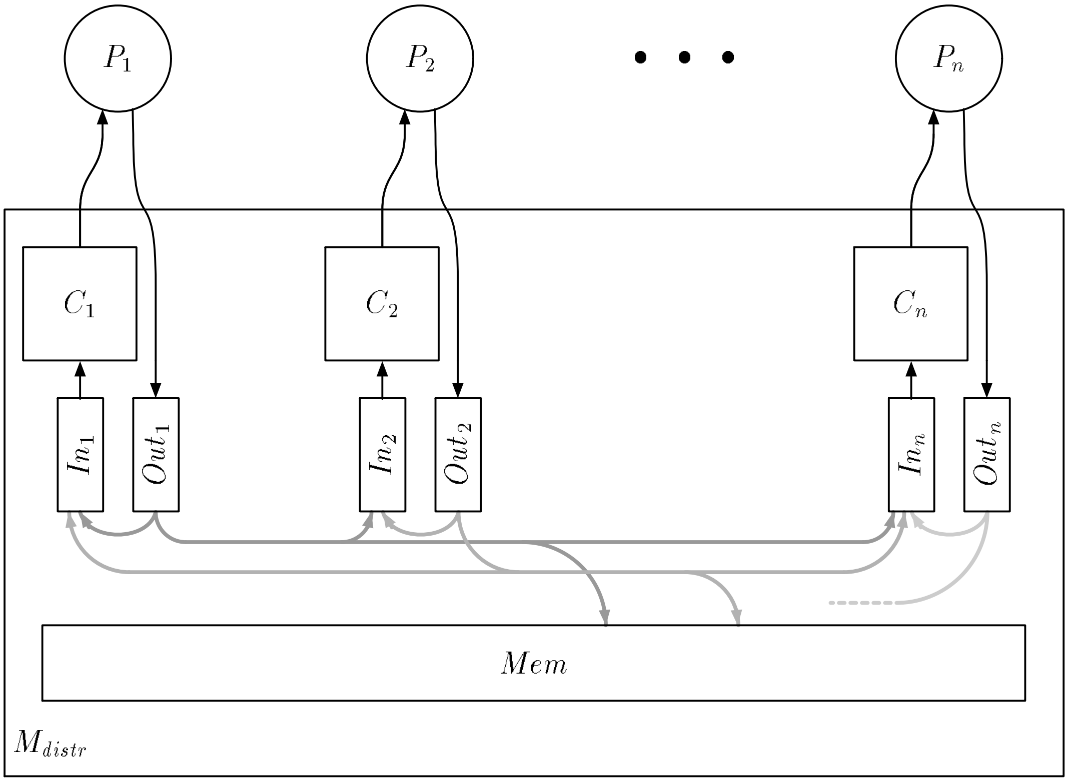

Figure 3: Architecture of M_{distr}

by an internal cache update event $\mathsf{CU}_j(d, a)$ as a result of which cache memory C_j gets updated. Cache misses are modeled by internal cache invalidate events: CI_i can arbitrarily remove locations from cache C_i. Caches are filled both as the delayed result of write events as well as through internal memory-read events, $\mathsf{MR}_i(d, a)$. The latter events intend to model the effect of a cache-miss: in that case the read event suspends until the location is copied from memory.

A read event $\mathsf{R}_i(d, a)$, predictably, stalls until a copy of location a is present in C_i but also until the copy contains a correct value in the following sense: sequential consistency demands that a processor P_i reads the value at a location a that was most recently written by P_i unless some other processor updated a in the mean time. Hence, a read event $\mathsf{R}_i(d, a)$ cannot occur unless all pending writes in Out_i are processed as well as the cache update requests from In_i that correspond to writes of P_i. For this reason, such cache update request are marked (with a $*$).

The transition system in Figure 4 makes all this precise.

In this transition system caches are modeled as partial functions from the set of locations to the set of values. Cache update (CU) actions produce 'variant functions': $update(C_i, d, a)$ stands for the function f that coincides with C_i except 'at' a where $f(a) = d$. Cache

E	Event	Allowed if	Action
$\surd$	$\mathsf{R}_i(d,a)$	$C_i(a) = d \wedge Out_i = \{\}$ $\wedge$ no $*$-ed entries in In_i	
$\surd$	$\mathsf{W}_i(d,a)$		$Out_i := append(Out_i, (d,a))$
	$\mathsf{MW}_i(d,a)$	$head(Out_i) = (d,a)$	$Mem[a] := d$; $Out_i := tail(Out_i)$; $(\forall k \neq i \;::\; In_k := append(In_k, (d,a)))$; $In_i := append(In_i, (d,a,*))$
	$\mathsf{MR}_i(d,a)$	$Mem[a] = d$	$In_i := append(In_i, (d,a))$
	$\mathsf{CU}_i(d,a)$	$head(In_i)$ is either (d,a) or $(d,a,*)$	$In_i := tail(In_i)$; $C_i := update(C_i, d, a)$
	CI_i		$C_i := restrict(C_i)$

Initially: $\forall a \;\; Mem[a] = 0$
$\wedge \forall i = 1 \ldots n \;\; C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\}$

Fairness: no action other than CI_i can be always enabled but never taken

MW—memory write MR—memory read
CU—cache update CI—cache invalidate

Figure 4: M_{distr}

invalidate (CI) actions yield 'restrictions' of functions: $restrict(C_i)$ stands for any function whose domain is included in that of C_i and which coincides with C_i on its domain.

For M_{distr} there is a distinction between the external behavior, Beh(M_{distr}) and the *internal behavior*, IBeh(M_{distr}) that comprises the maximal sequences of internal and external events that M_{distr} can generate (obviously we have Beh(M_{serial}) = IBeh(M_{serial})). Observe that for $s \in$ IBeh(M_{distr}), $s \upharpoonright i$ denotes the subsequence of *external* read and write-events of P_i in s.

The interface of the serial memory and the caching protocol in [ABM93] differs from the one we use. There, a $\mathsf{R}_i(d,a)$-event in either protocol is split into an (input) event $\mathsf{ReadRequest}_i(d,a)$, which is always enabled, and an (output) event $\mathsf{ReadReturn}_i(d,a)$ that behaves as the $\mathsf{R}_i(d,a)$-event. The main reason for doing so is their use of I/O automata specifications in which input events must be always enabled. However, that paper also stipulates that a process i must not do otherwise than engage in a Return event after it has issued a Request. This means that the intended interface is synchronous so that not using I/O automata and having simple read and write external events seem to be the conceptually clearer approach.

References

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–206, 1993.

[BFKR92] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report SR-TR-9202001, Kendall Square Research, Boston, 1992.

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

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, pages 63–79, 1992.

[Mos93] D. Mosberger. Memory consistency models. *ACM SIGOP Operating Systems Review*, 27(1):18–27, 1993.