Using Timestamping and History Variables to Verify Sequential Consistency*

Tamarah Arons

The John von Neumann Minerva Center for Verification of Reactive Systems,
Weizmann Institute of Science, Rehovot, Israel
tamarah@wisdom.weizmann.ac.il

Abstract. In this paper we propose a methodology for verifying the sequential consistency of caching algorithms. The scheme combines timestamping and an auxiliary history table to construct a serial execution 'matching' any given execution of the algorithm. We believe that this approach is applicable to an interesting class of sequentially consistent algorithms in which the buffering of cache updates allows stale values to be read from cache. We illustrate this methodology by verifying the high level specifications of the lazy caching and ring algorithms.

In shared memory multiprocessor systems a *memory consistency model* specifies how memory operations will appear to execute to the programmer. The closer the memory consistency model forces the shared memory to behave as a *serial* memory system – a system in which all operations are performed atomically directly on memory with no buffering or caching (Figure 1(a)) – the easier it is for the programmer to write correct code for the system. However, the stricter the memory model the more hardware and compiler optimizations are disallowed. *Sequential consistency* is an intuitive memory model, in which, "the result of any execution is the same as if the [memory] operations 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 the program"[24]. Sequential consistency is a relatively restrictive model when compared with the more relaxed memory models (such as partial or total store ordering, or release consistency) which are supported by some commercially available architectures (e.g. PowerPC, SPARC, Digital Alpha)[1].

Many sequentially consistent models implement *coherence*, an even stricter consistency model. Whereas an execution is sequentially consistent if all of the processors' local views can be interleaved to form a single serial behavior, regardless of the relative ordering of events at different processors, coherence requires that the events, as ordered *globally*, be a trace of serial memory [2].

To prove sequential consistency of a proposed memory implementation M it suffices to construct, for every σ_M, an execution of M, a matching serial execution σ_S such that all operations in σ_S read and write the same values as in

* Research supported in part by a grant from the German-Israel bi-national GIF foundation and a gift from Intel.

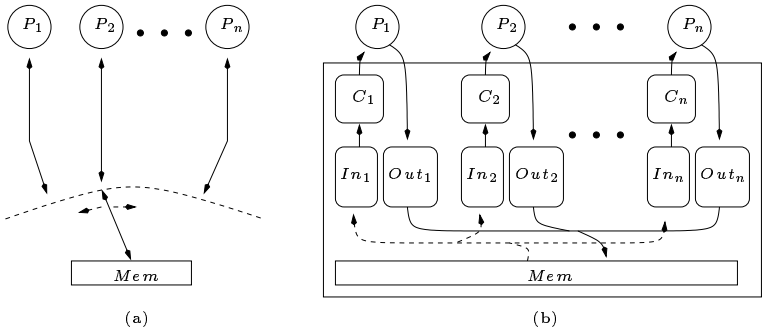

Fig. 1. Architecture of (a) a serial memory and (b) the lazy caching algorithm

σ_M. However, the creation of such a "witness" serial execution may require that a potentially unbounded number of operations be re-ordered. In fact, the problem of verifying sequential consistency is known to be undecidable [3]. Thus, unlike coherence which can often be verified quite easily, sequential consistency does not comfortably fit the pattern of standard refinement techniques (trace inclusion, bisimulation, testing preorder). The non-coherent lazy caching algorithm was therefore proposed by Rob Gerth as an example on which different refinement methods can be tried [15], and in 1999 a special edition of *Distributed Computing* was devoted to this project [13].

In this paper we present a proof methodology which involves *timestamping* the cache reads and shared memory updates of an execution and placing them in a *history table*. Intuitively, every processor P_i has a cache C_i which contains a subset of the values in the shared memory at some time $t_i \leq t_G$, where t_G is the global system time. All writes to memory occurring in the interval $(t_i, t_G]$ have not yet been applied to C_i. The *local time* t_i is precisely the time at which the global memory had contents consistent with C_i. We timestamp instructions with the local time (and other information, in order to create a total ordering between instructions executing at the same local time) and place them in a *history table* ordered by timestamp. The information in the history table contains sufficient information for a matching serial execution to be built, and the algorithm to be proved sequentially consistent.

We believe that this methodology is suitable for the verification of the sequential consistency of many non-coherent memory models, as demonstrated by our applying this proof method, using the PVS [27] theorem prover, to two examples, *lazy caching* [2, 15] and a *ring* algorithm [6][1]. While this methodology is theoretically applicable to coherent snoopy protocols, we believe that it is more complicated than is required for such algorithms. Current work considers increasing the automation of deductive proofs, and we hope later to consider the application of the methodology to other classes of caching algorithms.

[1] The PVS files are available at [4]

Event	Enabling conditions	Action
$\mathrm{R}_i(a,d)$	Instruction pc_i is "READ a"	$pc_i := pc_i + 1$
	$\wedge C_i(a).valid \wedge C_i(a).data = d$	
	$\wedge$ no *starred* entries in In_i	
	$\wedge Out_i = \{\}$	
$\mathrm{W}_i(a,d)$	Instruction pc_i is "WRITE a, d"	$Out_i := push(Out_i,(a,d)) \wedge pc_i := pc_i + 1$
$\mathrm{MW}_i(a,d)$	$head(Out_i) = (a,d)$	$Mem[a] := d \wedge Out_i := tail(Out_i)$
		$\wedge \forall_{k \neq i}\, In_k := push(In_k,(a,d))$
		$\wedge In_i := push(In_i,(a,d,*))$
$\mathrm{MR}_i(a)$	$C_i(a).valid = false$	$In_i := push(In_i,(a,Mem[a]))$
$\mathrm{CU}_i(a,d)$	$head(In_i) = (a,d) \vee$	$In_i := tail(In_i) \wedge C_i(a).data := d$
	$head(In_i) = (a,d,*)$	$\wedge C_i(a).valid := true$
$\mathrm{CI}_i(a)$	$C_i(a).valid = true$	$C_i(a).valid := false$
I_i (idle)		

Fig. 2. Lazy Caching transitions

The paper is structured as follows: In Section 1 we describe the lazy caching algorithm. In Section 2 we explain how timestamping and the history table are used to derive a serial execution. In Section 3 we define the ring algorithm and describe how it fitted into our methodology. Section 4 discusses related works and in Section 5 we summarize our conclusions.

1 Lazy Caching

The "lazy cache algorithm" [2] is a sequentially consistent protocol in which cache updates can be postponed, and writes are buffered, allowing processors to access stale cache data.

As illustrated in Figure 1(b), the system consists of n processors, $P_1, \ldots, P_n$ with each P_i owning a cache C_i, and FIFO *in-* and *out-queues* In_i and Out_i, respectively. We have further associated with each processor an unbounded *instruction list*, containing instruction of the form "READ a" and "WRITE a, d". Instructions in the instruction list are executed sequentially, with a program counter, pc_i, pointing to the next instruction.

A processor P_i initiates a write event W_i by placing a record recording the instruction address and new value at the tail of Out_i. When this record reaches the top of Out_i it can be popped off and the memory write MW_i occurs. That is, the shared memory is updated, and a new record recording the address and value is placed in the *in*-queue In_j of all processors P_j. The copy placed in In_i is *starred*. When the entry at the head of In_i is popped off a cache update CU_i occurs, and C_i is updated with the value recorded in the In_i entry.

A read event R_i can be performed if the address a requested is in the cache, Out_i is empty and In_i does not contain any starred entries. The value read is that in the cache. We note that this value may differ from that in the memory if a write to a is buffered in In_i. Locations (which are not currently in cache) can be

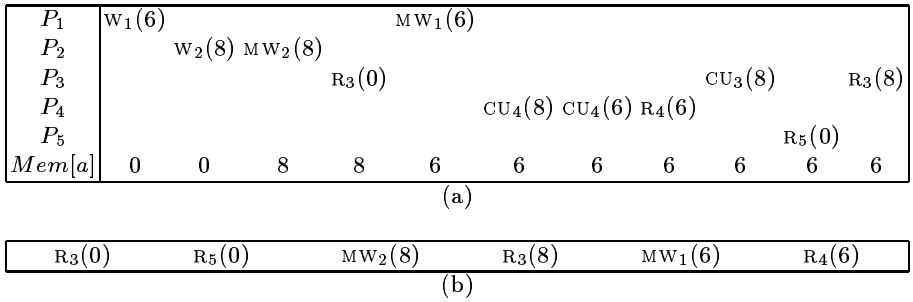

Fig. 3. (a) A partial execution of the lazy caching algorithm. All transitions refer to address a. Time increases from left to right. (b) A matching serial execution, where "read" and "write" instructions correspond to R and MW events.

brought into the cache by placing the memory value in the *in* queue in a memory read (MR$_i$) action, and can be summarily evicted by cache invalidation (CI).

In our interleaving model at any step a processor can either initiate a read or write (if one is enabled), pop an entry off its *in*- or *out*-queue if they are non-empty, initiate a cache update, invalidate a cache entry, or idle (I). The system is parameterized by the number of processors and there is no restriction on the maximum size of the queues, the address space, or the set of memory values. Our model, summarized in Figure 2, very closely resembles that of Gerth [15]. The reader is referred to this paper, or our PVS source files [4], for more information.

An example execution fragment In Figure 3(a) we consider a very small execution sequence which illustrates the non-coherent nature of the lazy caching algorithm. We assume that address a has initial value 0. Process P_1 initiates a write of 6 to a, placing the tuple $(a, 6)$ on its *out*-queue. Process P_2 then initiates a write of 8 to a. Process P_2 pops $(a, 8)$ off Out_2, in a memory write MW$_2$ action, pushing the (address, data) tuple onto the *in*-queues of all processors. Sometime thereafter action MW$_1$ also occurs. Process P_3 reads the value of 0 for a, updates its cache with 8, and then reads 8 as the value of a, while the write of 6 is buffered. Process P_4 updates its cache with both values before reading reading a as 6; process P_5 reads a as 0.

We note that the memory is updated in the opposite order to which the writes were initiated, and thus a has the final value of 6. Furthermore, processors P_3 and P_5 read stale values for a *after* P_4 has read the new value.

2 Creating a Serial Execution

To prove an algorithm sequentially consistent we show that each of its executions has an equivalent *serial* execution. In the serial execution all operations are executed directly on memory, in some sequential order, and the operations

of each individual processor are in program order, where "read" and "write" instructions correspond to R and MW events. It is shown that reads in the two executions return the same value, and the final memory values are identical. Figure 3(b) gives the serial execution corresponding to the lazy caching execution of Figure 3(a).

2.1 Logical time

Each processor has a view of memory which is consistent with the values memory had at some time in the past: It sees the memory as it was before it was modified by the last x writes, these being the writes which are buffered in the *in*-queue.

The *global time* t_G is determined by an auxiliary global clock, and is initially zero. Every time a memory write occurs the global time is incremented by one.

Each processor has an auxiliary local clock which counts the number of writes which have been applied to its cache. This clock gives its *local time*. It is updated each time a process performs a cache update which was initiated by a memory write. These cache updates are termed *countable*. (In order to distinguish countable cache updates from those initiated by memory reads, we add an auxiliary processor id field to *in*-queue records. An entry is the result of a memory read exactly if the processor id in the record is that of the processor and the record is not starred.) The processor has a view of memory consistent with the values that memory held when the global time was the current local time of the processor.

Every read (R) or memory write (MW) event in the system is given a unique *timestamp* when it occurs. The timestamp is a tuple (t, r, id), where t is the local time at which the event occurs, r is the numbers of reads which this processor has performed since the last counted cached update, and id is the identifier of the processor that initiated the read/write. On a read $\mathrm{R}_i(a, d)$ we add to the history table H an entry $\mathrm{R}_i(a, d)$, its timestamp $(t_i, r_i + 1, i)$ and the current program counter, pc_i of P_i. The local read counter, r_i, is incremented by 1. On a memory write $\mathrm{MW}_j(a, d)$ we add to the history table H an entry $\mathrm{MW}_j(a, d)$, its timestamp $(t_G + 1, 0, j)$ and pc_j and we set $t_G := t_G + 1$. On a counted cache update CU_k we set $t_k := t_k + 1, r_k := 0$.

The timestamps induce a strict order on memory events:
$(t_1, r_1, id_1) \prec (t_2, r_2, id_2) \Leftrightarrow t_1 < t_2 \vee t_1 = t_2 \wedge (r_1 < r_2 \vee r_1 = r_2 \wedge id_1 < id_2)$

Time 0 is the time given to all reads of the initial, unmodified memory. For every $t_i > 0$ the "smallest" timestamp with time t_i will always be a memory write (MW), as the *reads* field of a timestamp is zero exactly when it represents a memory write operation. Since the local clocks are incremented every time that a cache update is performed, there is only one memory write at time t_i and all other operations timestamped with $t = t_i$ are reads. As they are all reads from the same memory, with no intervening writes, they will return the same value irrespective of the ordering between them. However, it is desirable that the program order of each processor be maintained, and this is done by the *reads* field of the timestamp. The *id* field of the timestamp is used to order operations at the same local time by different processors. The relative ordering of these operations is unimportant, and ours in one of a number of possibilities.

(a)

Instruction	Action	Timestamp (t, r, id)	P_1 $t\ a\ r$	P_2 $t\ a\ r$	P_3 $t\ a\ r$	P_4 $t\ a\ r$	P_5 $t\ a\ r$	Global Time	Memory a
			0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0	0
P_1 : a := 6	$W_1(a, 6)$		0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0	0
P_2 : a := 8	$W_2(a, 8)$		0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0	0
	$MW_2(a, 8)$	$(1, 0, 2)$	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	1	8
P_3 : read a	$R_3(a, 0)$	$(0, 1, 3)$	0 0 0	0 0 0	0 0 1	0 0 0	0 0 0	1	8
	$MW_1(a, 6)$	$(2, 0, 1)$	0 0 0	0 0 0	0 0 1	0 0 0	0 0 0	2	6
	$CU_4(a, 8)$		0 0 0	0 0 0	0 0 1	1 8 0	0 0 0	2	6
	$CU_4(a, 6)$		0 0 0	0 0 0	0 0 1	2 6 0	0 0 0	2	6
P_4 : read a	$R_4(a, 6)$	$(2, 1, 4)$	0 0 0	0 0 0	0 0 1	2 6 1	0 0 0	2	6
	$CU_3(a, 8)$		0 0 0	0 0 0	1 8 0	2 6 1	0 0 0	2	6
P_5 : read a	$R_5(a, 0)$	$(0, 1, 5)$	0 0 0	0 0 0	1 8 0	2 6 1	0 0 0	2	6
P_3 : read a	$R_3(a, 8)$	$(1, 1, 3)$	0 0 0	0 0 0	1 8 1	2 6 1	0 0 1	2	6

(b)

Index	Timestamp	Operation	pc
1	$(0, 1, 3)$	$R_3(a, 0)$	1
2	$(0, 1, 5)$	$R_5(a, 0)$	1
3	$(1, 0, 2)$	$MW_2(a, 8)$	1
4	$(1, 1, 3)$	$R_3(a, 8)$	2
5	$(2, 0, 1)$	$MW_1(a, 6)$	1
6	$(2, 1, 4)$	$R_4(a, 6)$	1

(c)

Instruction	$Mem[a]$
P_3 : read a	0
P_5 : read a	0
P_2 : a := 8	8
P_3 : read a	8
P_1 : a := 6	6
P_4 : read a	6

Fig. 4. An execution of the lazy caching algorithm with history table and matching serial execution. (a) Building the history table. (b) The history table ordered by timestamp. (c) A serial execution.

These counters and timestamps are variants of Lamport clocks [23]. However, in our system each processor updates its clock independently, without reading the timestamps on incoming messages.

2.2 Extracting a serial execution from the history table

The history table is an ordered list of entries sorted in non-decreasing order of timestamp. Since memory writes always have a greater timestamp than any other elements in the table at the time they occur they are appended to its end. Reads, however, may be inserted in the middle of the history table. The function $size(H)$ returns the number of entries in H. For every $x \leq size(H)$, $H[x]$ refers to the x'th entry of H.

In Figure 4(a) we revisit the example of Section 1, showing how the history table would be constructed. For each processor the table records its local time t, the value it stores for a, and r, the number of reads it has performed since the last countable cache update. The timestamp column indicates the timestamp of the entry which is added to the history table at the step in which it is added. Time progresses from top to bottom in the table.

A serial execution can be derived from the history table such that the i'th entry in the history table corresponds to the i'th operation in the serial execution. It is proved that in this serial execution every processor issues its instructions in the same order as in the original execution, all reads return the same values as in the lazy caching execution, and the final memory values are the same as in the original execution.

In Figure 4(b) we present the history table built in the example of Figure 4(a), with entries ordered by timestamp. The table illustrates all the fields in the history table. Figure 4(c) illustrates the serial execution which is derived.

2.3 The proof

The auxiliary *history* (H) list and *memHist* array and *readValues* arrays are intrinsic to the presented proof. Each processor has a *readValues* array which maps instruction indices to values. Every time a read operation occurs the value read is stored in the relevant entry of the *readValues* array. This array is later used to insure that the lazy caching and serial executions return identical values for every read. The *memHist* array is a history of memory contents, where $memHist[t]$ is a copy of the shared memory at global time t. In addition, $memHist$ also stores for every time t the processor id and program counter for the instruction that updated memory from $memHist[t-1]$ to $memHist[t]$. We also found it useful to add auxiliary fields to the *in-* and *out-*queue entries: in addition to the address, value and "*" fields, we added auxiliary fields recording the processor id and program counter of the related instruction, and the global time at which the related event occurs. We note that this time field is *not* used to update the processors local clocks, or any other variables. Some of the data structures are detailed in Figure 5.

In order to construct the serial execution we prove a one to one relationship between executed operations and history table entries. The bulk of the proof effort involved manually defining properties of the lazy caching algorithm and then proving their invariance in the PVS[27] system. We list some of the invariants used in the proof.

For every two entries $H[x]$ and $H[y]$ of history table H with timestamps (t_x, r_x, id_x) and (t_y, r_y, id_y) respectively, and $x, y \leq size(H)$:

- If $x \neq y$ then $(t_x, r_x, id_x) \neq (t_y, r_y, id_y)$. (Distinct entries have distinct timestamps).
- $x < y$ iff $(t_x, r_x, id_x) \prec (t_y, r_y, id_y)$. ($H$ is ordered by timestamp).
- Entry $H[x]$ corresponds to a memory write operation iff $r_x = 0$.
- If $t_x = t_y$ and $r_x = 0$ then $r_y \neq 0$. (At most one memory write at any global time).
- For all $0 < t \leq t_G$ there is an index $z \leq size(H)$ such that $H[z]$ is timestamped $(t, 0, id)$ for some id. (Every time period greater than zero is initiated by a memory write).
- For all $0 < r < r_x$, there is an entry $H[z]$, $z < x$ timestamped (t_x, r, id_x) in H. (Reads are counted sequentially, with no gaps in the counting).

Type	Definition
TIME	$\mathbb{N}$
PROC_ID	$1 \ldots n$; for $n > 1$ a system parameter
ADDRESS	$\mathbb{N}$
VALUE	$\Re$
PC_RANGE	$\mathbb{N}^+$
MEMORY	ADDRESS $\mapsto$ VALUE

Basic types

Field	Type
t	TIME
r	$\mathbb{N}$
id	PROC_ID
operation	{R, MW}
address	ADDRESS
data	VALUE
pc	PC_RANGE

Entries of the history table, H

Field	Type
memory	MEMORY
id	PROC_ID
pc	PC_RANGE

Entries of $memHist$

Field	Type
cache	ADDRESS $\mapsto$ [$valid$: BOOLEAN, $data$: VALUE]
pc	PC_RANGE
inQueue, outQueue	QUEUE
t	TIME
$readCounter$	$\mathbb{N}$
$readValues$	PC_RANGE $\mapsto$ VALUE

Processors of the lazy caching system

Field	Type
address	ADDRESS
star	BOOLEAN
data	VALUE
t	TIME
pc	PC_RANGE
id	PROC_ID

Queue entries

Fig. 5. Some of the data structures. Auxiliary variables in the processor and queue structures are italicized.

- The time t_x is not greater than the global time t_G and if t_x is greater than the local time t_{id_x} then there is an entry in In_{id_x} corresponding to $H[x]$.
- The contents of $memHist$ for the current global time equal the current memory. That is, $memHist[t_G] = Mem$.
- For every address a and processor P_i with cache C_i and local time t_i, $C_i(a).valid \rightarrow C_i(a).data = memHist[t_i](a)$. The values of locations in the cache match the $memHist$ values for the processor's local time.
- For every occupied entry $In_i[k]$ of In_i, $t_i \leq In_i[k].t \leq t_G$ and if $t_i = In_i[k].t$ then $In_i[k]$ records a non-countable cache update. Intuitively, for every t such that $t_i < t \leq t_G$ there is an In_i-entry which will be used to update t_i.
- The program counter $H[x].pc$ is less than pc_{id_x}.
- For every value pc less than the program counter pc_i of P_i either there is an entry $H[z], z \leq size(H)$ with timestamp (t_z, r_z, i) such that $H[z].pc = pc$, or there is an entry of Out_i corresponding to this instruction.
- The value $P_{id_x}.readValues[H[x].pc] = memHist[t_x](a)$ where a is the address in the pc'th instruction of P_{id_x}. (The values in the $readValues$ array match the $memHist$ values for the time of the transition.)

The serial execution is inductively built in a list S where $S[x].mem$ and $S[x].procs$ give the global memory and processor states in the serial system after x execution steps. Intuitively, the x'th entry of S corresponds to the x'th entry of H, for all $x \leq size(H)$. That is, in the serial execution transitions occur in the order in which they appear in the history table.

We now define predicate α which describes the relationship between the lazy caching data structures L and S. For clarity we prefix data structures in the lazy caching algorithm with L where confusion could arise.

1. The first entry, $S[0]$, fulfills the initial conditions of the serial system.
2. For every $0 \leq x < size[H]$, $\rho_{serial}(S[x], S[x+1])$. That is, there is a transition in the serial system from $S[x]$ to $S[x+1]$.
3. For every $0 \leq x \leq size[H]$, $S[x].mem = L.memHist[H[x].t]$. That is, the global memory at the x'th entry in S matches the memory recorded in $L.memHist$ for time $H[x].t$.
4. For every processor P_i and program index p, if the p'th instruction of P_i is a read instruction then $S[size[H]].readValues[i,p] = L.readValues[i,p]$. That is, every read in the two systems returns the same value.
5. The program counter of processor P_i at the end of the sequential execution, $S[size(H)].pc_i$, is equal to $L.pc_i$ if $L.Out_i$ is empty, and the (auxiliary) program counter field in the top $L.Out_i$ entry, otherwise.

We prove inductively that for every reachable lazy caching state L there is an S such that $\alpha(L,S)$: We first prove that predicate α holds for the initial states of the two systems, and then that if $\alpha(L,S)$ holds, then for any L' such that $\rho_{lazy}(L,L')$ is a lazy caching transition, we can build an S' such that $\alpha(L',S')$.

From parts (1) and (2) of α S records a legal serial execution. Given that $L.memHist[t_G]$ is proved to equal $L.Mem$, the currently lazy caching memory, from (3) we can deduce that the memory values in the two systems agree. From (4) we prove that both systems return the same value for every read.

We complete the proof by showing that the lazy caching system can always progress meaningfully.

3 The Ring Algorithm

In order to test the applicability of our methodology we applied it also to a model based on Collier's ring algorithm [6]:

Processors $P_0, \ldots, P_{n-1}$ are connected in a ring, with P_i sending messages only to its successor, $P_{i+1 \bmod n}$. The channels between every two successive processors are FIFO queues of messages. Processor P_0 is designated the *supervisor*. If processor $P_i, i \neq 0$ wants to perform a write of value v to address a it sends to its successor a *WriteRequest(a, v)* message and enters a *waiting* state. This write request is passed around the ring until it reaches the supervisor. The supervisor updates memory with this address and value, and then sends a *WriteReturn(a, v)* message. On receiving a *WriteReturn* message all processors update their caches, and then pass it on to their successor. Process P_i also releases itself from its *waiting* state and can proceed. When the write return reaches the supervisor, it is removed from the system.

A processor can execute a read instruction if the address is in its cache. Otherwise it sends a *ReadRequest*, which the supervisor answers with a *ReadReturn*. After thus bringing the address into the cache, the read can be executed.

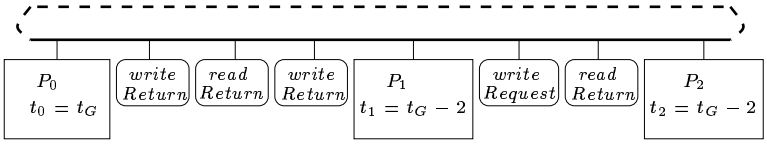

Fig. 6. An example configuration in the ring algorithm

The supervisor accesses memory directly (its local cache *is* the "shared memory") and never issues *ReadRequest* or *WriteRequest* messages. On performing a write it sends a *WriteReturn* message so that all other caches can be updated.

This model fits neatly into our framework. As in the lazy caching example, cache reads and updates to the shared memory are entered into the history table when they occur. (In this algorithm the memory update occurs when a *WriteReturn* in initiated by the supervisor.) The supervisor increments its local clock when it sends a *WriteReturn*, and all other processors increment their local clocks on receiving the *WriteReturn*. The local time of the supervisor is the global system time. The local time t_i of P_i is the global time minus the number of *writeReturns* on channels between P_0 and P_i. An example configuration is given in Figure 6.

4 Related Works

Various methodologies, ranging from CSP [5, 9], to abstraction [16] and model checking [19] have been used to verify lazy caching. The primary difficulty in verifying lazy caching seems to be that at the time that a memory is updated by a write in the lazy caching system, it is not known how many reads reading the stale value will still occur. That is, nondeterministic choices in the abstract (serial) system occur earlier than in the concrete (lazy caching) system. One solution is to input the computation of the concrete system into a transducer, which queues segments of the concrete computation until they can be matched with an abstract execution [21]. Similarly, [19] propose a finite state observer that observes and re-orders the memory operations, while [22] use an auxiliary queue to record writes which have updated memory but have not yet updated the cache. Step-wise refinement, in which the lazy caching system is transformed in a number of steps to a serial system, is used in [5] and [22]. Composition [20] and abstraction [16] are two other methodologies proposed, while in [9] decomposition is coupled with the use of CSP to prove trace inclusion.

The paper introducing lazy caching [2] presents a semantic proof that it is sequentially consistent. A *WriteCounter* is used to assign a sequence numbers to updates of the shared memory. Reads are assigned numbers according to the last write which the processor has popped off its *in*-queue. An auxiliary $Hist$ variable is used, with semantics similar to that of our $memHist$ variable.

Of the above mentioned verification efforts only [19] has been mechanized at all. The model-checking verification in [19] is of a restricted system in which

there is no *out*-queue and the *in*-queue is of size at most one. Given the problems of state explosion, it is unclear how a more detailed system could be verified. It is claimed that the type of abstractions that are used in [16] could be computed algorithmically, thus partially mechanizing this proof.

Timestamping, using variants of logical Lamport clocks [23], has been used to verify various memory consistency models [7, 8]. The algorithms are verified at a lower level than we have considered, including message passing protocols. Timestamping is used to divide logical time into *coherence epochs*, intervals of logical time in which a node has read-only or read-write access to a block of data. Thus, it is possible for one epoch to contain multiple, or no, stores. Furthermore the same write can be given different timestamps when it is used to update different caches. In contrast, in our timestamping each memory update is identified with an epoch and has a unique timestamp. This underscores a difference in our approaches to memory consistency – whether block control or memory contents are the primary concern. The difference in emphasis is appropriate given the different levels (high level versus message passing) at which verification occurs, and the different algorithms considered. The proofs presented are entirely manual.

Theorem proving has been used by Park and Dill [11, 12] and Stoy et al [28] to verify cache coherence protocols at the message passing level. Park and Dill *aggregate* the steps of each transaction in the implementation into a single atomic transition in the specification. A *commit* point is identified, for each transition, and the aggregation function intuitively is a function completing committed instructions. This methodology has been used to effectively verify a detailed model of the complex FLASH protocol. However, it is unclear how it could be used in our examples, where instructions may commit out of order (a read instruction may return an older value than a previous read, by another processor, for the same address). In [28] a PVS [27] implementation of Lamport's TLA [25] is used. Queues are *drained* to empty them of messages, and an abstraction function used to show refinement between two protocols.

A lot of research has been done on using model checking to verify cache coherence protocols. However, due to the difficulties of verifying large systems many of these methodologies are restricted. E.g., the 'test model-checking' of [17] in incomplete, the work by Delzanno, Pong and Dubois [10, 14] based on FSMs is only appropriate to coherent algorithms. Lazic [26] shows that data independence theorems can be used to make model checking of cache protocols more tractable.

Our construction of a serial execution is reminiscent of work by Glusman and Katz [18]. They allow independent operations to be re-ordered to create a *convenient computation*. Our "convenient" serial execution is not only a re-ordering of the events, but also a change in the nature of the occurring events.

There are more points of similarity between our work and those mentioned above. The auxiliary variables in [22, 19] perform some of the functions of our history table. While timestamping has been used previously in verifying cache consistency protocols [8], the similarities between this work and ours are in the terminology more than the semantics. Our timestamping is closer in meaning to the *WriteCounter* variable in [2]. Their *Hist* variable is also similar to

our *memHist* variable. However, the proof in [2] is 'on a semantical level and not grounded in a refinement methodology'[15]. By creating a full timestamping scheme, and using a history table, we have developed a formal verification framework which allows mechanical verification, and can easily be applied to different verification problems.

The centrality of the history table, and the method in which it is coupled with timestamping is new, and provides a relatively simple proof which is amenable to *mechanical verification*. We believe that mechanical verification provides a higher degree of confidence than pen and paper proofs, and testifies to a relatively simple and natural methodology.

5 Conclusion

In this paper we present a refinement methodology for the verification of sequential consistency. Given that the general problem is known to be undecidable, our proof method cannot be complete. However, we believe that there is a class of 'difficult', non-coherent algorithms, to which this methodology is suited, as illustrated by the successful verification of the lazy caching and ring algorithms.

We take cache reads and shared memory updates to be the important events to be recorded, and show that a correct ordering of these events allow the construction of a matching serial execution. While the idea of using timestamps (or, more generally, Lamport clocks) to order events is far from new, the timestamping that we have devised is particularly well suited to sequential consistency. It allows us to give a relative order (timestamp) to an "important event", when it occurs, relative to all past and possible *future* such events in the system. The history table provides a means of dynamically ordering these events, so that a serial execution can be extracted.

The methodology is sound – when it is applied a corresponding serial execution can be built. Since all steps are mechanically verified in the PVS theorem prover, this gives a very solid proof of sequential consistency.

The major drawback of this methodology is the large amount of human effort required (several person-weeks), devoted primarily to deriving the invaraint properties and directing the theorem prover. We are currently researching techniques to increase the automation of the proofs, and hope later to consider the extension of our methodology to other classes of algorithms.

Acknowledgements: Prof. Amir Pnueli, my supervisor, provided invaluable criticsims and suggestions; Jürgen Niehaus suggested the ring algorithm.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 9512, Rice University, 1995.
2. Y. Afek, G. Brown, and M. Merrit. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.

3. R. Alur, K. L. McMillan, and D. Peled. Model checking of correctness conditions for concurrent objects. In *MICS'96*:219–228, 1996.
4. T. Arons. Homepage. `http://www.wisdom.weizmann.ac.il/~tamarah/caching/`.
5. E. Brinksma. Cache consistency by design. *Dist. Comp.*, 12:61–74, 1999.
6. W. W. Collier. *Reasoning about Parallel Architectures.* Prentice Hall, 1992.
7. A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin. Lamport clocks: Reasoning about shared-memory correctness. Technical Report CS-TR-1367, University of Wisconsin, Madison, 1998.
8. A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin. Lamport clocks: Verifying a directory cache-coherence protocol. In *Proc. 10th ACM Symp. Parallel Algorithms and Architectures (SPAA)*, 1998.
9. J. Davies and G. Lowe. Using CSP to verify sequential consistency. *Dist. Comp.*, 12:91–103, 1999.
10. G. Delzanno. Automatic verification of parametrized cache coherence protocols. *CAV'00*:53–68, 2000.
11. D. L. Dill and S. Park. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *SPAA'96*:288–296, 1996.
12. D. L. Dill and S. Park. Verification of cache coherence protocols by aggregation of distributed transactions. In *Theory of Computing Systems.* 1998.
13. *Distributed Computing*, Volume 12 Number 2/3, 1999.
14. M. Dubois and F. Pong. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.
15. R. Gerth. Sequential consistency and the lazy caching algorithm. *Dist. Comp.*, 12:57–59, 1999.
16. S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Dist. Comp.*, 12:75–90, 1999.
17. R. Ghughal, G. Gopalakrishnan, A. Mokkedem, and R. Nalumasu. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. *CAV'98*:464–376, 1998.
18. M. Glusman and S. Katz. Mechanizing proofs of computation equivalence. *CAV'99*:354–367, 1999.
19. T. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. *CAV'99*:301–315, 1999.
20. W. Janssen, M. Poel, and J. Zwiers. The compositional approach to sequential consistency and lazy caching. *Dist. Comp.*, 12:105–127, 1999.
21. R. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. *Dist. Comp.*, 12:129–149, 1999.
22. P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Dist. Comp.*, 12:151–174, 1999.
23. L. Lamport. Time, clocks and the ordering of events. *Communications of the ACM*, 21(7):558–565, 1978.
24. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-82(9):690–691, 1979.
25. L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Sys.*, 16(3):872–923, May 1994.
26. R. S. Lazic. *A Sematic Study of Data Independed with Appliations to Model Checking.* PhD thesis, Oxford University Computing Laboratory, 1999.
27. S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. *TPCD'94*:258–279, 1994.
28. J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Formal Methods Europe, FME'01*, Springer-Verlag, 2001.