Distrib. Comput. (1999) 12: 91–103

DISTRIBUTED COMPUTING

Using CSP to verify sequential consistency

Gavin Lowe[1], **Jim Davies**[2]

[1] Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK
[2] Programming Research Group, Oxford University, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Summary. This paper shows how the theory of Communicating Sequential Processes (CSP) can be used to establish that a protocol guarantees sequential consistency. The protocol in question is an accepted design based upon lazy caching; it is an ideal example for the comparison of different formal description techniques.

Key words: Sequential consistency – Lazy caching protocol – CSP – Specification – Verification

1 Introduction

In shared-memory multiprocessor systems, the time taken to perform memory access operations is critical. In most designs, this is reduced by equipping each processor with a cache: a local image of the shared store. If each cache contains a copy of the locations that the corresponding processor is most likely to access, then any delay due to shared memory access will be minimised.

However, if a system contains multiple copies of the same data, care must be exercised if the system is to behave in a predictable and satisfactory fashion. Whenever a processor updates some location, any caches that contain a copy of that location must be updated to match. This is the rôle of a consistency protocol.

Many consistency protocols operate by marking other copies of the data as invalid, so that subsequent access requires a read from shared memory. This marking must be done immediately: no further reads or writes can occur until all caches have been marked. In highly-distributed multiprocessor systems, the delay caused by such atomic 'write and mark' operations is unacceptable. Such systems require a more relaxed view of data consistency.

In [Lam79], Lamport introduced the notion of *sequential consistency*:

The result of any execution is the same as if the 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 its program.

This notion is now widely employed in multiprocessor designs.

In [ABM93] the authors propose an algorithm – the *lazy caching protocol* – for ensuring sequential consistency. In this paper, we present a rigorous verification of the lazy caching protocol, using the language of Communicating Sequential Processes (CSP). By reasoning about the interaction between processors, caches, and shared memory, we are able to confirm that any sequence of read and write actions performed by the lazy caching protocol is sequentially consistent.

The description of the protocol and the ensuing verification serve to illustrate the essential features of CSP: the process language, the theory of observations, and the theory of refinement. The result is an interesting case study in specification and analysis: an ideal basis for comparison with other formal description techniques.

The paper begins with a brief introduction to CSP. In Sect. 2, we explain the use of the process language to represent communicating systems, and the rôle of the traces model in reasoning about interaction. We then review the description of the lazy caching protocol, originally presented in [ABM93] and [Ger95], to prepare the reader for our formal description.

In Sect. 4, we use the abstraction of [Ger95] to produce a formal characterisation of sequentially-consistent behaviour. We then produce a process representation of the protocol design. Each component is represented as a sequential process whose behaviour depends upon a small amount of state information. These processes are then combined to produce a description of the complete system.

The verification of the protocol design is described in Sects. 6 and 7. We begin by showing that each component of the protocol exhibits a local version of sequential consistency; we are then able to show that this is enough to guarantee that the entire system is sequentially consistent.

In Sect. 8, we show how each component can be rewritten as a parallel combination of caching and queueing processes. We are able to demonstrate that this implementation exhibits the same behaviour as the original, sequential design. This completes the verification of the protocol. The paper ends with a discussion of the issues raised.

2 Communicating Sequential Processes

The theory of Communicating Sequential Processes (CSP) is a mathematical approach to the design and analysis of distributed systems. It has undergone a steady evolution, encompassing real-time [Dav93] and probabilistic [Low93] theories of process behaviour, although the definitive account is still that of [Hoa85].

2.1 Processes and events

In applications of CSP, each component of a system is represented by a *process* – an abstract program – which records the points at which certain events can take place. These events are atomic, instantaneous synchronisations. They represent interaction between system components, or between a system and its environment.

The simplest of all processes is $Stop$: this can perform no events, and represents a component that is unable to interact. We add events to a process description using the prefix operator: if P is a process, then $a \rightarrow P$ is another process, which is able to perform a before behaving as P.

More often that not, several alternative behaviours are possible. If P and Q are both processes, then $P \Box Q$ is a process that is prepared to behave either as P or as Q. The choice is resolved by the first event to occur: for example, if this event is performed by P, then the subsequent behaviour is that of P.

The availability of an alternative may depend upon the state of the process concerned. If B is a boolean condition (whose truth depends upon the parameters of the process and the values received in inputs), then $\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is a process that will behave as P if B is true, and behave as Q otherwise.

The interaction between conditional processes and choice is essential to our style of process definition. Consider the choice process

$$\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Stop$$
$$\Box$$
$$\mathsf{if}\ B\ \mathsf{then}\ Q\ \mathsf{else}\ Stop\ .$$

This describes a choice of processes in which one or both alternatives may be unavailable. A genuine choice between P and Q is offered only if both A and B are true. Since $Stop$ is unable to resolve a choice – it can perform no events – the above process is equivalent to the following nested conditional:

$$\mathsf{if}\ A\ \wedge \neg\ B\ \mathsf{then}\ P$$
$$\mathsf{else\ if}\ B \wedge \neg\ A\ \mathsf{then}\ Q$$
$$\mathsf{else\ if}\ A \wedge B\ \mathsf{then}\ P \,\Box\, Q$$
$$\mathsf{else}\ Stop\ .$$

In general, if the behaviour when the guard is false is that of $Stop$, we will omit the 'else' clause from the expression, writing, for example,

$$\mathsf{if}\ A\ \mathsf{then}\ P$$
$$\Box$$
$$\mathsf{if}\ B\ \mathsf{then}\ Q\ .$$

This process structure is reminiscent of the Language of Guarded Commands [Dij76]. Indeed, we refer to the boolean expressions themselves as 'guards'. However, it is important to remember that the choice is resolved by the first *event* to occur: the truth of the corresponding guard is not enough.

We may use compound events to represent the passing of data values from one process to another. The set of compound events $\{c.v \mid v \in Value\}$ may be used to describe communication of values of type $Value$ on channel c. The process $c!v \rightarrow P$ is ready to output the value v; the process $c?x \rightarrow P$ is ready to input any value x on the same channel. The ! and ? symbols are not primitive operators, but they may be defined by abbreviation:

$$c!v \rightarrow P \mathrel{\widehat{=}} c.v \rightarrow P\ ,$$
$$c?x \rightarrow P \mathrel{\widehat{=}} \Box_{x \in Value}\, c.x \rightarrow P\ ,$$

using a possibly-infinite indexed form of the choice construct. The ! and ? symbols may be mixed within a single prefixing construct. For example, when we come to consider the lazy caching protocol, we will represent a read of value d from address a of cache i by an event $R.i.a.d$. A processor that reads the value currently held in a particular address a would have a definition of the form $R!i!a?d \rightarrow \ldots$, representing that the processor is willing to accept any value d from the memory. Conversely, the cache itself will allow any address a to be read, but will insist that the value read matches the current contents of a; if the contents of a is given by $cache(a)$, then the cache will have a definition with a clause of the form $R!i?a!cache(a) \rightarrow \ldots$.

We write $P \parallel Q$ to denote the parallel combination of P and Q. Whenever a process appears in such a combination, it is associated with a set of events: its *alphabet*. Any event from this set will require the cooperation of the process concerned. For example, if processes P and Q are associated with alphabets αP and αQ, then their parallel combination can perform:

- an event a from $\alpha P \setminus \alpha Q$ exactly when P can perform a;
- an event b from $\alpha Q \setminus \alpha P$ exactly when Q can perform b;
- an event c from $\alpha P \cap \alpha Q$ exactly when both P and Q can perform c.

A shared event acts as a synchronisation between two or more processes; it occurs simultaneously for every process involved.

Shared events are not concealed within a parallel combination: it is possible that three or more processes may be involved in a single synchronisation. In an indexed parallel combination, such as $\|_{i \in I}\, P_i$, any or all of the components may share a single event.

If an event is local to a combination of processes, then we may conceal it from the rest of the system using the hiding operator. We write $P \setminus A$ to denote the result of concealing events from set A within process P. This process may perform such events internally, but they are no longer available for synchronisation with other components.

Finally, processes are introduced by equations which may include recursive instances of the process name. For example:

$P \mathrel{\widehat=} a \rightarrow P$

defines a process P which behaves as '$a \rightarrow P$'.

2.2 Traces

Each term in the language of processes is associated with a set of *traces*. This set describes the various sequences of interaction that are possible for the process concerned. Each trace is a finite sequence of events: for example, the process

$a \rightarrow b \rightarrow Stop$

is associated with the traces $\langle\rangle$, $\langle a\rangle$, and $\langle a, b\rangle$.

The empty trace/sequence $\langle\rangle$ is a trace of every process: it represents a history in which no events are performed. The event prefix operator introduces the possibility of an event: if P is associated with a set of traces S, then $a \rightarrow P$ may engage in any trace from the set

$\{\langle a\rangle^\frown s \mid s \in S\}$

obtained by prefixing each trace in S with an a. There is also the possibility that no events are performed, so we may infer that

$$traces[\![a \rightarrow P]\!] \;=\; \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in traces[\![P]\!]\} \;.$$

Similar arguments apply to the other process constructors in our language. The result is a denotational semantic model in which the meaning of each process is its trace set.

2.3 Specification

Constraints upon trace sets may be used to express requirements upon process behaviour. For example, suppose that a process P is capable of performing – amongst other events – both of the events a and b. If we wish to specify that P never performs an a after a b, then we have only to insist that in any trace of process P, the event a never appears after b. Formally:

$$\forall tr_1, tr_2 \bullet tr = tr_1^\frown \langle b\rangle^\frown tr_2 \Rightarrow tr_2 \upharpoonright \{a\} = \langle\rangle \;.$$

Here and henceforth the variable tr has a special status, representing an arbitrary trace of the process in question. The above predicate states that if tr contains a b event, then the part of tr following the b contains no occurrences of event a. To express this condition, we have used the trace projection operator $\upharpoonright$, which restricts the trace to the chosen set (here $\{a\}$).

We will require several other operators for our trace specifications:

- head, tail and last: if s is a non-empty trace, then head s is the first event in s, tail s is the part of the trace that follows the first event, and last s is the last event to appear.
- prefix: if s and t are both traces, then s prefix t is true exactly when s is a prefix of t: that is, when there is some trace u such that $s^\frown u = t$.
- $\setminus$: if s is a trace and A is a set of events, then $s \setminus A$ is the trace obtained by removing all instances of events from A. This operation complements trace projection, in that

  $$s \setminus A \;=\; s \upharpoonright (\Sigma \setminus A) \;,$$

  where Σ is the set of all events.
- $\Downarrow$: if s is a trace, then $s \Downarrow c$ denotes the sequence of values passed on channel c: for example, if s were the trace

  $$\langle a,\ a,\ c.1,\ d.4,\ a,\ c.3,\ d.5,\ c.2\rangle \;,$$

  then $s \Downarrow c$ would be the sequence $\langle 1,\ 3,\ 2\rangle$. This may be extended in the obvious way to reveal the sequence of data values passed on a set of channels.

Since our traces are finite sequences, it is a simple matter to provide formal definitions of these operators.

In general, we will wish to project a trace onto a set of compound events, or to consider the values passed on a collection of channels. If A is a set of event or channel names, then we write $c.A$ to denote the set $\{c.a \mid a \in A\}$, consisting of all possible compound events of the form $c.a$ for $a \in A$. Similarly, we write $A.d$ to denote the set $\{a.d \mid a \in A\}$. This convention will be used extensively in this paper.

To establish that a process meets a trace specification, we must show that every trace of the process satisfies the corresponding predicate. We define

$$P \textbf{ sat } S(tr) \mathrel{\widehat=} \forall\, tr \in traces[\![P]\!] \bullet S(tr) \;.$$

Although the **sat** is a relation between process syntax and predicates, it may be seen as a refinement relation. If we identify the process P with its set of traces, and the predicate S with its characteristic set, then the statement above asserts that every trace of P is a trace of S, or P refines S.

2.4 Equivalence and refinement

The traces model supports a theory of refinement and equivalence between processes. A process Q is a refinement of another process P if and only if every trace of Q is a possible trace of P:

$$P \sqsubseteq Q \mathrel{\widehat=} traces[\![Q]\!] \subseteq traces[\![P]\!] \;.$$

If this is the case, then Q is a *safe* replacement for P, in the sense that Q cannot engage in any behaviour that is not also possible for P.

If the refinement holds in both directions, then we say that the two processes are trace-equivalent:

$$P \equiv Q \mathrel{\widehat=} traces[\![P]\!] = traces[\![Q]\!] \;.$$

If this is the case, then P and Q may be used interchangeably. This leads to a number of simple algebraic laws for rewriting processes into equivalent forms: for example, the law

$$P \,\Box\, Stop \equiv P$$

shows how an empty choice may be removed from a menu. A complete set of algebraic laws for refinement and equivalence is included in [Hoa85, Bro83].

One law that we will require in this paper is an *expansion theorem*; it shows how a parallel combination may be rewritten as a choice of sequential evolutions:

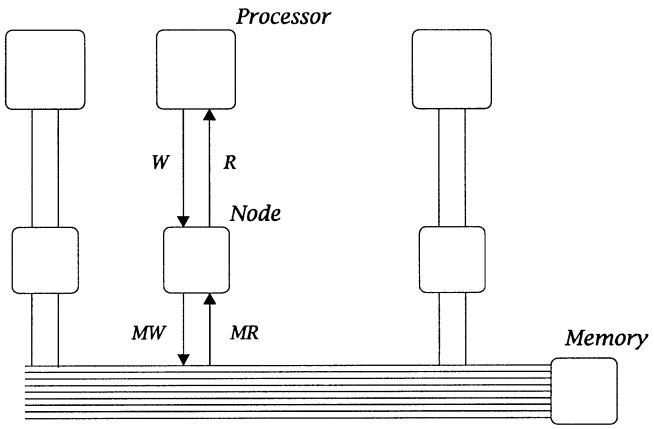

Fig. 1. The lazy caching protocol

Let $P = \Box_{x \in A}\ x \rightarrow P(x)$ and $Q = \Box_{y \in B}\ y \rightarrow Q(y)$. Then

$$P \parallel Q = \Box_{z \in C}\ z \rightarrow (P'(z) \parallel Q'(z)) \ ,$$

where:

$$C = (A \cap B) \cup (A \setminus \alpha Q) \cup (B \setminus \alpha P) \ ,$$

$$P'(z) = \text{if } z \in A \text{ then } P(z) \text{ else } P \ ,$$

$$Q'(z) = \text{if } z \in B \text{ then } Q(z) \text{ else } Q \ .$$

A parallel combination may be seen as a choice of initial events. The consequence of each event depends upon the components involved.

We will also require a result which states when any collection of mutually-recursive equations defines a unique process. Hence if two processes P and Q satisfy the same set of equations, they must be equivalent. For example, if we know that

$$P = a \rightarrow P' \ , \qquad P' = b \rightarrow P \ ,$$

and also that

$$Q = a \rightarrow Q' \ , \qquad Q' = b \rightarrow Q \ ,$$

then we may be sure that P and Q are the same process. This result holds whenever the right-hand side of each equation is *guarded*: that is, every instance of a recursive call is prefixed by an event. A more detailed discussion of guarded-ness can be found in [Dav93].

3 The lazy caching protocol

The lazy caching protocol sits between a collection of processors and a shared memory. It consists of a number of caching nodes, one for each processor: see Fig. 1. The nodes are connected via some form of bus architecture, which also links them to a shared memory. Each node may write to and read from the shared memory; a memory read is a private action, but a memory write will be observed by all of the other nodes: it will eventually lead to the local states of all other nodes being updated accordingly.

To better explain the behaviour of the protocol, we write:

- $R.i.a.d$ to represent a read by processor i of value d from address a;
- $W.i.a.d$ to represent a write by processor i of value d to address a.

Here and henceforth, we take i to be a member of the set $Index$ of process indices, a to be a member of the set $Address$ of memory addresses, and d to be a member of the set $Data$ of data values. The above events are interactions between the caching nodes and the corresponding processors. There are also internal communications between the caching nodes and the shared memory. We write:

- $MR.i.a.d$ to represent a memory read by caching node i of value d from address a;
- $MW.i.a.d$ to represent a memory write by caching node i of value d to address a. This memory write will also be seen by other nodes, and eventually lead to their local states being updated accordingly.

These interactions are not visible to the processors in the system.

When a processor writes to memory, represented by an event $W.i.a.d$, the new value is queued in the caching node until it can be written to the shared memory itself, represented by an event $MW.i.a.d$. The first event is a private communication between processor and caching node; the second will be observed by all caching nodes and the shared memory. Each node i is equipped with an *out queue* of pending memory writes: each (a, d) pair in the queue represents a pending $MW.i.a.d$ event;

Communications between the caching nodes and the shared memory do not have an immediate effect upon local copies. The effect of each communication is queued until the local copy – the cache – can be updated. We write:

- $CU.i.j.a.d$ to represent the updating at node i of the cache address a with value d, where the update was caused by a memory write at node j

Each node i is equipped with an *in queue* for pending cache updates: each (j, a, d) triple in the queue represents a pending $CU.i.j.a.d$ event.

If the first element in the in queue of cache i is of the form (a, d), then an $MW.i.a.d$ event may occur; the effect of this event is to remove this pair from i's in queue, and to add the triple (i, a, d) to every node's out queue.

If a particular address is not present in the local cache, then its value may be read from the shared memory. The effect of a memory read event is to add the triple $(0, a, d)$ to the in queue: the tag 0 represents that this update originates from the shared memory, rather than one of the caching nodes.

The queued communications may have an effect upon the behaviour of a caching node. A node will block any read of the cache by its local processor if either: (1) there are any memory writes pending: that is, if out is not empty; or (2) the effect of a local memory write has yet to propagate to the local cache: that is, if in holds a locally generated update, of the form (i, a, d) where i is the index of the local node. Provided that the queues maintain the order of the messages they hold, this protocol is enough to guarantee sequential consistency.

Finally, we must recognize that the cache will not, in general, hold a complete copy of the shared memory: some addresses may be removed from the local cache. We write:

Event	Allowed if	Action
$R.i.a.d$	$cache_i(a) = d \wedge out_i = \langle\rangle \wedge$ there are no locally generated updates in in_i	
$W.i.a.d$		$out_i := out_i \frown \langle(a, d)\rangle$
$MW.i.a.d$	head $out_i = (a, d)$	$memory := memory \oplus \{a \mapsto d\}$; $out_i :=$ tail out_i; $in_k := in_k \frown \langle(i, a, d)\rangle$, for each k
$MR.i.a.d$	$memory(a) = d$	$in_i := in_i \frown \langle(0, a, d)\rangle$
$CU.i.j.a.d$	head $in_i = (j, a, d)$	$in_i :=$ tail in_i; $cache_i := cache_i \oplus \{a \mapsto d\}$
$CI.i.a$		$cache_i := \{a\} \lhd cache_i$

Fig. 2. The lazy caching protocol

– $CI.i.a$ to represent the invalidation or removal of address a from the cache at node i.

The effect of each event is summarized in Fig. 2 (adapted from [Ger95]). The state of the shared memory is given by the function $memory$. Each node i has local cache given by $cache_i$, in queue in_i, and out queue out_i. When describing memory updates, we follow the Z notation [Spi87], writing $memory \oplus \{a \mapsto d\}$ to represent the memory function updated with address a set to value d. We write $\{a\} \lhd cache$ for the function obtained by removing a from the domain of the function $cache$.

Note that in the original presentation of the lazy caching algorithm [ABM93], the elements of the in queue did not include the index j, representing the node that generated this update; instead, locally-generated updates were flagged with a star. This change is designed to simplify the proof.

4 Sequential consistency

If we consider the combination of caching nodes and shared memory as a CSP process, then we may cast the sequential consistency property as a trace specification. We need only concern ourselves with events of the form $R.i.a.d$ and $W.i.a.d$, as these describe the service provided by the protocol at its external interface.

Below we will define a predicate $Serial(tr)$, which holds exactly when tr is a trace of a serial memory, and a predicate $Consistent(tr, tr')$, which holds exactly when the two traces tr and tr' agree upon the order of reads and writes as seen from each node. Then we can capture sequential consistency as a trace specification:

$$SC(tr) \mathrel{\widehat=} \exists\, tr' \bullet Serial(tr') \wedge Consistent(tr, tr') \ .$$

That is, tr is a trace of a sequentially consistent memory exactly when there is another trace tr' which is a trace of a serial memory, and consistent with tr.

A memory is said to be *serial* if the value obtained by reading from any address a matches the last value written to a. If we have a trace, then the last value written to address a may be determined by looking for the last event from the set $W.Index.a.Data$ to appear in the trace. We define a simple projection function to do this:

$$\begin{aligned}&\mathsf{last\ write}\,(a, tr) \mathrel{\widehat=}\\ &\quad \mathsf{last}\,(tr \Downarrow W.Index.a), \mathsf{if}\, tr \Downarrow W.Index.a \neq \langle\rangle\\ &\quad \mathsf{initial}\,(a), \qquad \text{otherwise.}\end{aligned}$$

The last value written to a in trace tr is the last value passed on the set of channels $W.Index.a$ during tr. If nothing has been written to a, the function has a special value $\mathsf{initial}\,(a)$.

We may then define the predicate $Serial$ on traces as follows:

$$\begin{aligned}Serial\,(tr) \mathrel{\widehat=} \forall\, tr_0, i, a, d \bullet tr_0 \frown \langle R.i.a.d\rangle \ \mathsf{prefix}\ tr \Rightarrow\\ \mathsf{last\ write}\,(a, tr_0) = d \ .\end{aligned}$$

That is, if a read event of address a occurs after trace tr_0, then the data value d that is read must be the last value written to a.

To define the second of our two predicates, we must consider the external interface provided by each caching node. This is described by the set of events $External_i$, where

$$External_i \mathrel{\widehat=} W.i.Address.Data \cup R.i.Address.Data \ .$$

Two traces are consistent if they agree upon the order of external events at each node: that is

$$\begin{aligned}&Consistent\,(tr, tr') \mathrel{\widehat=}\\ &\quad \forall\, i : Index \bullet tr \upharpoonright External_i = tr' \upharpoonright External_i \ .\end{aligned}$$

Consistent traces may differ in the relative order of external communications at different nodes.

As an example, consider the following trace of a sequentially consistent memory system:

$$\langle W.3.x.0,\ W.1.x.1,\ W.2.y.2,\ R.3.y.2,\ R.3.x.0,\ R.3.x.1\rangle \ ,$$

that is: node 3 writes $x := 0$; node 1 writes $x := 1$; node 2 writes $y := 2$; node 3 reads $y = 2$; node 3 reads $x = 0$; and node 3 reads $x = 1$. Note that node 3 is able to read $x = 0$ after node 1 has written $x := 1$; the effect of this write has clearly not yet propagated throughout the system.

The above trace is consistent with the following trace, which could be performed by a serial memory system:

$$\langle W.3.x.0,\ W.2.y.2,\ R.3.y.2,\ R.3.x.0,\ W.1.x.1,\ R.3.x.1\rangle \ .$$

In this trace, the order of cause and effect is more apparent. The value of x is updated between the first and second read events for node 3.

In practice, still weaker notions of consistency may be adopted. For example, the following trace is possible under processor consistency [Mos93]:

$$\langle W.1.x.1,\ W.2.x.2,\ R.3.x.1,\ R.4.x.2,\ R.3.x.2,\ R.4.x.1\rangle \ ,$$

but is outlawed under sequential consistency. The effects of the write operations have arrived at nodes 3 and 4 in a different order; there can be no serial equivalent of this trace.

5 Formal design

We will now construct a CSP description of the protocol design, using a single process to represent each of the caching nodes. The process representing node i has the alphabet

$$\begin{aligned}\alpha Node_i \mathrel{\widehat=} &W.i.Address.Data \cup R.i.Address.Data \cup\\ &MW.Index.Address.Data \cup MR.i.Address.Data \cup\\ &CU.i.Index.Address.Data \cup CI.i.Address \ .\end{aligned}$$

Note that $Node_i$'s alphabet includes all $MW.j$ events; for $j \neq i$, this represents $Node_i$ receiving from $Node_j$ notification of a write originating at $Node_j$.

The process representing node i may communicate with its processor on each of the channels $R.i$ and $W.i$, and with the rest of the memory system on channels $MW.Index$ and $MR.i$. The cache update events $CU.i.Index.Address.Data$ and cache invalidation events $CU.i.Address$ will not be shared.

The state of a node process has three components, representing an input queue of pending cache updates, an output queue of pending memory writes, and a cache of data. These are:

- in, a sequence of triples from the product $Index \times Address \times Data$;
- out, a sequence of pairs from the product $Address \times Data$;
- $cache$, a partial function from $Address$ to $Data$.

The protocol allows events of the form $R.i.a.d$ only when the input queue contains no locally-generated updates. It is useful to define a predicate that is true precisely in this case:

$$\mathsf{no\text{-}local}_i\,(in) \mathrel{\widehat{=}} in \upharpoonright (\{i\} \times Address \times Data) = \langle\rangle\,.$$

This predicate is true if and only if none of the indices in the queue match the local index i.

A node process is then defined by the following family of equations:

$$\begin{aligned}
&Node_i\,(in, out, cache) \mathrel{\widehat{=}}\\
&\quad \mathsf{if}\ out = \langle\rangle \wedge \mathsf{no\text{-}local}_i\,(in)\\
&\quad \mathsf{then}\ R.i?a!cache(a) \rightarrow Node(in, out, cache)\\
&\quad \Box\\
&\quad MR.i?a?d \rightarrow Node(in^\frown\langle(0, a, d)\rangle, out, cache)\\
&\quad \Box\\
&\quad W.i?a?d \rightarrow Node(in, out^\frown\langle(a, d)\rangle, cache)\\
&\quad \Box\\
&\quad \mathsf{if}\ in \neq \langle\rangle\\
&\quad \mathsf{then}\ CU.i.j.a.d \rightarrow Node(\mathsf{tail}\ in, out, cache \oplus \{a \mapsto d\})\\
&\qquad \text{where } (j, a, d) = \mathsf{head}\ in\\
&\quad \Box\\
&\quad \mathsf{if}\ out \neq \langle\rangle\\
&\quad \mathsf{then}\ MW.i.a.d \rightarrow Node(in^\frown\langle(i, a, d)\rangle, \mathsf{tail}\ out, cache)\\
&\qquad \text{where } (a, d) = \mathsf{head}\ out\\
&\quad \Box\\
&\quad \Box_{j \neq i}\ MW.j?a?d \rightarrow Node(in^\frown\langle(j, a, d)\rangle, out, cache)\\
&\quad \Box\\
&\quad \Box_{a \in \mathrm{dom}\, cache}\ CI.i.a \rightarrow Node(in, out, \{a\} \lhd cache)\,.
\end{aligned}$$

Considering the seven alternatives in order of appearance:

- If the out queue is empty and the in queue contains no updates originating from this node, then the process may engage in any event of the form $R.i.a.cache(a)$. There are no restrictions upon the address a – it may be seen as an input from the processor – but once a is chosen, the value is fixed by the current state of the cache – it may be seen as an output.
- The node process is always ready to read any value d from any address a of the shared memory. The effect is to add the triple $(0, a, d)$ to the in queue.
- The process is always ready to accept a write request from the processor. This has the effect of adding an appropriate pair to the out queue.
- If the input queue is non-empty, then the cache may be updated accordingly. If the head of the input queue is the triple (j, a, d), then this has the effect of over-riding the cache function with the maplet $a \mapsto d$.
- If there is a memory write pending, then it may be performed. The first element is removed from the out queue, broadcast to the rest of the system, and added to the in queue, tagged with this node's identity.
- The process is always ready to observe a memory write event from another node. This event adds an appropriate triple to the end of the input queue, recording the address, the data value, and the identity of the node responsible.
- The process is always willing to drop an address a from the local cache. We do not model the circumstances under which this occurs.

In the initial state, both queues are empty, and the cache is in an initial state described by the function 'initial', mentioned earlier:

$$Node_i\,(\langle\rangle, \langle\rangle, \mathsf{initial})\,.$$

The protocol makes use of a shared memory, which we may also represent as a CSP process. The alphabet of the shared memory is given by:

$$\begin{aligned}
\alpha Memory \mathrel{\widehat{=}}\ & MW.Index.Address.Data\\
& \cup\, MR.Index.Address.Data\,.
\end{aligned}$$

The process representing the shared memory is parameterized by a function $memory$ from $Address$ to $Data$:

$$\begin{aligned}
&Memory(memory) \mathrel{\widehat{=}}\\
&\quad MW?i?a?d \rightarrow Memory(memory \oplus \{a \mapsto d\})\\
&\qquad \Box\\
&\quad MR?i?a!memory(a) \rightarrow Memory(memory)\,.
\end{aligned}$$

The memory may accept memory write events from any node, and update its state accordingly, and may allow any memory read events with the appropriate data value.

The protocol itself is described by the parallel combination of all of the node processes and the shared memory:

$$\begin{aligned}
&Protocol \mathrel{\widehat{=}}\\
&\quad Memory\,(\mathsf{initial}) \parallel \Big\|_{i \in Index} Node_i\,(\langle\rangle, \langle\rangle, \mathsf{initial})\,.
\end{aligned}$$

Each node process will cooperate on events from $\alpha Node_i$; the memory cooperates on events from $\alpha Memory$.

6 Component properties

To show that the protocol behaves according to our formal characterisation of sequential consistency, we begin by examining the behaviour of each node. We exhibit a simple and obvious property that is satisfied by each node, and

then show that this implies a less obvious property, which we term *local sequential consistency*. Later, we will show that if all the nodes satisfy local sequential consistency, then the complete system is sequentially consistent.

When an address a is read at node i, the value read will be the last value corresponding to a cache update of a at i. This value is extracted from a trace using the function 'last update$_i$', defined as follows:

$$\begin{array}{l}\mathsf{last\ update}_i\,(a, tr, cache) \mathrel{\widehat{=}} \\ \quad \mathsf{last}\,(tr \Downarrow CU.i.Index.a), \\ \qquad\qquad\qquad \text{if } tr \Downarrow CU.i.Index.a \neq \langle\rangle \\ \quad cache(a), \\ \qquad\qquad\qquad \text{otherwise.}\end{array}$$

As with last write, an initial value is provided if the address has never been updated, which is obtained using the function $cache$.

We now define a function that returns the sequence of $Index.Address.Data$ triples added to the in queue of node i during a trace tr; we define the function using a sequence comprehension:

$$\begin{array}{l}\mathsf{ins}_i(tr) \mathrel{\widehat{=}} \\ \quad \langle j.a.d | c.a.d \leftarrow tr, c = MW.j \vee c = MR.i \wedge j = 0\rangle\ .\end{array}$$

For each $MW.j.a.d$ event, the triple (j, a, d) is added to the in queue; for each $MR.i.a.d$ event, the triple $0.a.d$ is added to the in queue.

We claim that the node process in its initial state,

$$Node_i\,(\langle\rangle, \langle\rangle, \mathsf{initial})\ ,$$

satisfies the following specification:

$$tr \Downarrow CU.i \text{ prefix } \mathsf{ins}_i\,(tr) \tag{1}$$
$$\wedge$$
$$tr \Downarrow MW.i \text{ prefix } tr \Downarrow W.i \tag{2}$$
$$\wedge$$
$$\begin{array}{l}\forall\, tr_0, a, d \bullet tr_0 \frown \langle R.i.a.d\rangle \text{ prefix } tr \ \Rightarrow \\ \qquad tr_0 \Downarrow MW.i = tr_0 \Downarrow W.i \wedge \qquad (3) \\ \qquad tr_0 \Downarrow CU.i.i = tr_0 \Downarrow MW.i \wedge \qquad (4) \\ \qquad \mathsf{last\ update}_i\,(a, tr_0, \mathsf{initial}) = d\ . \qquad (5)\end{array}$$

The variable in acts as a queue, taking input from the MW and $MR.i$ channels, and outputting to $CU.i$; so the values transmitted over $CU.i$ (a sequence of $Index.Address.Data$ triples) is a prefix of the values added to the queue as a result of MW or $MR.i$ events (Equation 1). The variable out also acts as a queue, inputting from $W.i$ and outputting to $MW.i$; so the values transmitted over $MW.i$ is a prefix of the values transmitted over $W.i$ (Equation 2). Finally, if a read of address a occurs after trace tr_0, that is, $tr_0 \frown \langle R.i.a.d\rangle$ prefix tr, then:

- the out queue must be empty, so the values transmitted over $MW.i$ in trace tr_0 must be the same as the values transmitted over $W.i$ (Equation 3);
- the in queue must contain no locally-generated updates, so the cache updates originating at i (the $CU.i.i$ events) must be the same as the memory writes originating at i (the $MW.i$ events) (Equation 4);
- the value read should be correct, that is, it should be the value corresponding to the last cache update of address a at this node (Equation 5).

This property is a consequence of the general node process,

$$Node_i\,(in, out, cache)\ ,$$

satisfying the following property:

$$\begin{array}{l}tr \Downarrow CU.i \text{ prefix } in \frown \mathsf{ins}_i\,(tr) \\ \wedge \\ tr \Downarrow MW.i \text{ prefix } out \frown (tr \Downarrow W.i) \\ \wedge \\ \forall\, tr_0 \bullet tr_0 \frown \langle R.i.a.d\rangle \text{ prefix } tr \Rightarrow \\ \qquad tr_0 \Downarrow MW.i = out \frown (tr_0 \Downarrow W.i) \wedge \\ \qquad tr_0 \Downarrow CU.i.i = (in \Downarrow i) \frown (tr_0 \Downarrow MW.i) \wedge \\ \qquad \mathsf{last\ update}_i\,(a, tr_0, cache) = d\ ,\end{array}$$

where $in \Downarrow i$ represents the sequence of $Address.Data$ pairs with index i currently held in the in queue. This is straightforward – but lengthy – to prove; we give part of the proof in the appendix to this paper.

6.1 Local sequential consistency

We now identify the essential property of a node that allows us to establish that the protocol is sequentially consistent. We say that a trace is *locally serial* at node i if the values read from the cache agree with the values written into its in queue. That is, whenever address a is read at node i, the value obtained is the same as that last written to a in the shared memory:

$$\begin{array}{l}Locally\_Serial_i\,(tr) \mathrel{\widehat{=}} \\ \quad \forall\, tr_0, a, d \bullet tr_0 \frown \langle R.i.a.d\rangle \text{ prefix } tr \Rightarrow \\ \qquad\qquad\qquad \mathsf{last\ in}_i\,(a, tr_0) = d\ ,\end{array}$$

where $\mathsf{last\ in}_i\,(a, tr_0)$ gives the last value corresponding to address a added to the in queue, or the initial value of a:

$$\begin{array}{l}\mathsf{last\ in}_i\,(a, tr) \mathrel{\widehat{=}} \\ \quad \mathsf{last}\,(\mathsf{ins}_i\,(tr) \Downarrow Index.a), \quad \text{if } \mathsf{ins}_i\,(tr) \Downarrow Index.a \neq \langle\rangle \\ \qquad \mathsf{initial}\,(a), \qquad\qquad\qquad\qquad \text{otherwise.}\end{array}$$

Later, we will use the fact that the shared memory is serial to deduce that the value $\mathsf{last\ in}_i(a, tr)$ is the same as the last value written to address a in the shared memory:

$$\mathsf{last\ in}_i\,(a, tr) = \mathsf{last\ mwrite}\,(a, tr)\ ,$$

where:

$$\begin{array}{l}\mathsf{last\ mwrite}\,(a, tr) \mathrel{\widehat{=}} \\ \quad \mathsf{last}\,(tr \Downarrow MW.Index.a), \quad \text{if } tr \Downarrow MW.Index.a \neq \langle\rangle \\ \qquad \mathsf{initial}\,(a), \qquad\qquad\qquad\qquad \text{otherwise.}\end{array}$$

It is useful to define a predicate that is true of a trace precisely when:

- every $MW.i$ event is immediately preceded by a corresponding $W.i$ event; *and*

– every $W.i$ event is either immediately followed by the corresponding $MW.i$ event, or is followed only by other W events.

$$
\begin{aligned}
&W.i \text{ precedes } MW.i\ (tr) \mathrel{\widehat{=}} \\
&\quad \forall tr_0, a, d \bullet tr_0 \frown \langle MW.i.a.d \rangle \text{ prefix } tr \Rightarrow \\
&\qquad\qquad \text{last } tr_0 = W.i.a.d \\
&\quad \wedge \\
&\quad \forall tr_0, tr_1, a, d \bullet tr_0 \frown \langle W.i.a.d \rangle \frown tr_1 = tr \Rightarrow \\
&\qquad\qquad \text{head } tr_1 = MW.i.a.d \\
&\qquad\qquad \vee tr_1 \setminus W = \langle\rangle \ .
\end{aligned}
$$

If this predicate holds, then each write $W.i.a.d$ either has an immediate effect upon the shared memory – through a corresponding $MW.i.a.d$ event – or it has no effect and appears in a block of similar writes at the end of the trace.

We say that two traces tr and tr' are *locally consistent* if: they agree on the order of external events; they agree on the order of internal events; and each $MW.i$ in tr' is preceded by a corresponding $W.i$. Formally:

$$
\begin{aligned}
&Locally\_Consistent_i\ (tr, tr') \mathrel{\widehat{=}} \\
&\quad tr' \upharpoonright External_i = tr \upharpoonright External_i \wedge \\
&\quad\ tr' \upharpoonright Internal_i = tr \upharpoonright Internal_i \wedge \\
&\quad\ W.i \text{ precedes } MW.i\ (tr') \ .
\end{aligned}
$$

where

$$
\begin{aligned}
&Internal_i \mathrel{\widehat{=}} \\
&\quad MW.Index.Address.Data \cup MR.i.Address.Data \cup \\
&\quad\ CU.i.Index.Address.Data \cup CI.i.Address \ .
\end{aligned}
$$

That is, the set $Internal_i$ consists of those events at node i that do not form part of the external interface.

We define a trace to be *locally sequentially consistent* if it is locally consistent with a locally serial trace:

$$
\begin{aligned}
LSC_i\ (tr) \mathrel{\widehat{=}} \exists\, tr' \bullet\ & Locally\_Serial_i\ (tr') \\
& \wedge\ Locally\_Consistent_i\ (tr, tr') \ .
\end{aligned}
$$

6.2 Proof of local sequential consistency

We are now ready to prove that every trace of a node process must be locally sequentially consistent. That is,

$Node_i$ **sat** $LSC_i\ (tr)$.

We do this by showing that local sequential consistency is implied by the properties (1–5) of the node processes that we identified above.

Given a trace tr of $Node_i$, we produce a locally-serial trace tr' such that tr and tr' are locally-consistent. Informally, reads are moved to the position where the value read is consistent with the shared memory; and writes are moved to the position at which they have an effect upon the shared memory. More precisely:

– We move all $R.i$ events forward (towards the start of the trace) to a position just after the last MW or $MR.i$ event whose corresponding $CU.i$ event occurs before the original position of the $R.i$; this is the last MW or $MR.i$ event that has had an effect on the cache before the read (there must be such an event by property (1)). If there are several consecutive reads, without any intervening cache updates, then these read events keep their same relative order.
– We move all $W.i$ events backwards (towards the end of the trace) to a position just before the corresponding $MW.i$ event; if there is no corresponding $MW.i$ event then the $W.i$ is moved to the end of the trace, but with all such events keeping their same relative order.

These transformations are formalized in Appendix B, using a simple language of sequences.

This transformation satisfies a number of obvious properties. The internal events (those in $Internal_i$) keep the same order in each trace:

$$tr' \upharpoonright Internal_i = tr \upharpoonright Internal_i \ . \tag{6}$$

The read events are not reordered:

$$tr' \upharpoonright R.i = tr \upharpoonright R.i \ , \tag{7}$$

and neither are the writes:

$$tr' \upharpoonright W.i = tr \upharpoonright W.i \ . \tag{8}$$

Each W event is moved to precede the corresponding MW event:

$$W.i \text{ precedes } MW.i\ (tr') \ . \tag{9}$$

Each read event is moved to just after the last MW or $MR.i$ event whose corresponding $CU.i$ event has occurred; so the data passed on the $CU.i$ channel before the read in tr is the same as the data passed on the MW and $MR.i$ channels before the corresponding read in tr':

$$
\forall tr_0, tr_0' \bullet \begin{pmatrix} tr_0 \frown \langle R.i.a.d \rangle \text{ prefix } tr \\ \wedge\ tr_0' \frown \langle R.i.a.d \rangle \text{ prefix } tr' \\ \wedge\ tr_0 \upharpoonright R.i = tr_0' \upharpoonright R.i \end{pmatrix} \Rightarrow \\
tr_0 \Downarrow CU.i = \text{ins}_i\ (tr_0') \ . \tag{10}
$$

We will now show that trace tr' is locally serial, and that tr and tr' are locally consistent.

Local serialization. We need to show that any read event in tr' is consistent with the memory writes:

$$tr_0' \frown \langle R.i.a.d \rangle \text{ prefix } tr' \Rightarrow \text{last in}_i\ (a, tr_0') = d \ .$$

If $tr_0' \frown \langle R.i.a.d \rangle$ prefix tr' then by (7) there is some tr_0 such that

$$tr_0 \frown \langle R.i.a.d \rangle \text{ prefix } tr \wedge tr_0 \upharpoonright R.i = tr_0' \upharpoonright R.i \ . \tag{11}$$

So by (10):

$$tr_0 \Downarrow CU.i = \ \text{ins}_i\ (tr_0') \ . \tag{12}$$

Hence:

$$
\begin{aligned}
&\text{last in}_i\ (a, tr_0') \\
&= \{\text{definition of last in}_i\} \\
&\quad \begin{cases} \text{last}\,(\text{ins}_i\ (tr_0') \Downarrow Index.a), & \text{if ins}_i\ (tr_0') \Downarrow Index.a \neq \langle\rangle \\ \text{initial}\ (a), & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{array}{ll}
= \{\text{from (12)}\} & \\
\quad \mathsf{last}\,(tr_0 \Downarrow CU.i.Index.a), & \text{if } tr_0 \Downarrow CU.i.Index.a \neq \langle\rangle \\
\quad \mathsf{initial}\,(a), & \text{otherwise} \\
= \{\text{definition of last update}_i\} & \\
\quad \mathsf{last\ update}_i\,(a, tr_0, \mathsf{initial}) & \\
= \{\text{by (5) and (11)}\} & \\
\quad d\ . &
\end{array}
$$

Local-consistency. Most of the requirements for local consistency follow directly from Equations (6–9). It remains only to show that read and write events are not reordered with respect to one another. Suppose that corresponding reads are preceded by tr_0 in tr, and by tr'_0 in tr':

$$tr_0 ^\frown \langle R.i.a.d\rangle \,\mathsf{prefix}\, tr \wedge tr'_0 ^\frown \langle R.i.a.d\rangle \,\mathsf{prefix}\, tr' \\ \wedge\ tr_0 \upharpoonright R.i = tr'_0 \upharpoonright R.i\ .$$

We must show that the writes in tr_0 agree with those in tr'_0:

$$tr_0 \Downarrow W.i = tr'_0 \Downarrow W.i\ .$$

Now, from (10),

$$tr_0 \Downarrow CU.i = \mathsf{ins}_i\,(tr'_0)\ ,$$

so restricting to i ($i \neq 0$):

$$tr_0 \Downarrow CU.i.i = \ \mathsf{ins}_i\,(tr'_0) \Downarrow i = tr'_0 \Downarrow MW.i\ . \tag{13}$$

Also, from (9) and the definition of precedes:

$$tr'_0 \Downarrow MW.i = tr'_0 \Downarrow W.i\ . \tag{14}$$

Hence we have:

$$
\begin{array}{ll}
tr_0 \ \Downarrow\ W.i & \\
\quad = tr_0 \Downarrow MW.i & \text{[from (3)]} \\
\quad = tr_0 \Downarrow CU.i.i & \text{[from (3)]} \\
\quad = tr'_0 \Downarrow MW.i & \text{[from (13)]} \\
\quad = tr'_0 \Downarrow W.i & \text{[from (14)]}
\end{array}
$$

as required.

6.3 Shared memory properties

The shared memory $Memory(\mathsf{initial})$ is a serial memory that is written to on the MW channels, and read from the MR channels; therefore, if value d is read from address a, then d must be the last value written to a:

$$\forall\, tr_0, i, a, d \bullet tr_0 ^\frown \langle MR.i.a.d\rangle \,\mathsf{prefix}\, tr \ \Rightarrow \\ \mathsf{last\ mwrite}\,(a, tr_0) = d\ , \tag{15}$$

where last mwrite is as defined above.

This property is proved in a similar way to the properties of the node processes.

7 System properties

Consider a trace tr of the protocol. We will show that tr is a sequentially consistent trace: $SC(tr)$. Let tr_i be the projection of tr onto $\alpha Node_i$:

$$tr_i \mathrel{\widehat{=}} tr \upharpoonright \alpha Node_i\ . \tag{16}$$

Then from the definition of parallel composition, tr_i is a trace of $Node_i$, and so it is locally sequentially consistent. Hence there is some trace tr'_i such that

$$tr'_i \upharpoonright External_i = tr_i \upharpoonright External_i\ , \tag{17}$$

$$tr'_i \upharpoonright Internal_i = tr_i \upharpoonright Internal_i\ , \tag{18}$$

$$Locally\_Serial_i(tr'_i)\ , \tag{19}$$

$$W.i \ \mathsf{precedes}\ MW.i(tr'_i)\ . \tag{20}$$

Note also that $tr \upharpoonright \alpha Memory$ is a trace of the shared memory, and so satisfies equation (15). We use this fact to show that the functions last in and last mwrite return the same values when applied to traces of the system:

$$\forall\, tr \in traces\,[\![Protocol]\!];\ a \in Address; i \in Index \bullet \\ \mathsf{last\ in}_i\,(a, tr) = \mathsf{last\ mwrite}\,(a, tr)\ . \tag{21}$$

This is proven as follows:

$$
\begin{array}{ll}
\quad \mathsf{last\ in}_i\,(a, tr) & \\
= \{\text{definition of last in}_i\} & \\
\quad \mathsf{initial}\,(a), & \text{if } tr \upharpoonright (MW.Index.a \cup MR.i.a) = \langle\rangle \\
\quad d, & \text{if } \exists\, j \bullet \mathsf{last}\,(tr \upharpoonright (MW.Index.a \cup MR.i.a)) \\
& \qquad = MW.j.a.d \\
\quad d, & \text{if } \mathsf{last}\,(tr \upharpoonright (MW.Index.a \cup MR.i.a)) \\
& \quad = MR.i.a.d \\
= \{\text{definition of last mwrite; (15)}\} & \\
\quad \mathsf{last\ mwrite}\,(a, tr)\ . &
\end{array}
$$

We will produce a trace tr' such that tr' is a serial trace, and tr and tr' are consistent. Note that each of the tr'_i agree on the MW events:

$$tr'_i \upharpoonright MW = tr_i \upharpoonright MW = tr \upharpoonright MW = tr_j \upharpoonright MW \\ = tr'_j \upharpoonright MW\ .$$

We construct the trace tr' as follows:

- take the MW events from tr;
- for each i, insert the $R.i$, $MR.i$, $CU.i$ and $CI.i$ events so as to agree with the order given by tr'_i (these events do not appear in any other tr'_j for $i \neq j$, so this is always possible);
- insert the $W.i$ events before the corresponding $MW.i$, or at the end of the trace respecting the order given by tr'_i (again this is always possible because $W.i$ events occur only in tr'_i).

The resulting trace agrees with tr'_i on the order of events from $Node_i$, and has W events preceding the corresponding MW events, by construction:

$$\forall\, i \bullet tr' \upharpoonright \alpha Node_i = tr'_i\ , \tag{22}$$

$$\forall\, i \bullet W.i \ \mathsf{precedes}\ MW.i(tr')\ . \tag{23}$$

We will now prove that tr and tr' are consistent, and that tr' is a serial trace.

Consistency. We must show $\forall\, i \bullet tr' \upharpoonright External_i = tr \upharpoonright External_i$. This is a simple matter:

$$
\begin{array}{ll}
tr' \ \upharpoonright\ External_i & \\
\quad = tr'_i \upharpoonright External_i & \text{[from (22)]} \\
\quad = tr_i \upharpoonright External_i & \text{[from (17)]} \\
\quad = tr \upharpoonright External_i & \text{[from (16)]}
\end{array}
$$

Serialization. Note firstly that

$$tr \upharpoonright Internal_i = tr' \upharpoonright Internal_i \ . \tag{24}$$

The proof of this is very similar to the proof of consistency, and uses equation (18).

Suppose that a read occurs in tr' after trace tr'':

$$tr'' \frown \langle R.i.a.d \rangle \text{ prefix } tr' \ .$$

To show that tr' is a serial trace we must show that the read is of the appropriate value:

$$\text{last write}\,(a, tr'') = d \ .$$

Note that from (23) we have:

$$tr'' \Downarrow MW.Index = tr'' \Downarrow W.Index \ . \tag{25}$$

We proceed as follows:

$tr'' \frown \langle R.i.a.d \rangle$ prefix tr'

$\Rightarrow$ {property of $\upharpoonright$; from (24); traces of $Protocol$ are prefix closed}

$tr'' \upharpoonright \alpha Node_i \frown \langle R.i.a.d \rangle$ prefix $tr' \upharpoonright \alpha Node_i \wedge$
$\exists\, tr''' \in traces\,[\![Protocol]\!] \bullet$
$tr''' \upharpoonright Internal_i = tr'' \upharpoonright Internal_i$

$\Rightarrow$ {defining $tr_i'' \mathrel{\widehat{=}} tr'' \upharpoonright \alpha Node_i$; using (22)}

$tr_i'' \frown \langle R.i.a.d \rangle$ prefix $tr_i' \wedge$
$\exists\, tr''' \in traces\,[\![Protocol]\!] \bullet$
$tr''' \upharpoonright Internal_i = tr'' \upharpoonright Internal_i$
$= tr_i'' \upharpoonright Internal_i$

$\Rightarrow$ {(19); definition of $Locally\_Serial$}

$\text{last in}_i\,(a, tr_i'') = d \wedge$
$\exists\, tr''' \in traces\,[\![Protocol]\!] \bullet$
$tr''' \upharpoonright Internal_i = tr'' \upharpoonright Internal_i$
$= tr_i'' \upharpoonright Internal_i$

$\Rightarrow$ {definition of last in$_i$ depends only upon events in $Internal_i$}

$\exists\, tr''' \in traces\,[\![Protocol]\!] \bullet$
$\text{last in}_i\,(a, tr''') = d \wedge$
$tr''' \upharpoonright Internal_i = tr'' \upharpoonright Internal_i$

$\Rightarrow$ {from (21); $tr''' \in traces\,[\![Protocol]\!]$}

$\exists\, tr''' \in traces\,[\![Protocol]\!] \bullet$
$\text{last mwrite}(a, tr''') = d \wedge$
$tr''' \upharpoonright Internal_i = tr'' \upharpoonright Internal_i$

$\Rightarrow$ {definition of last mwrite depends only upon events in $Internal_i$}

$\text{last mwrite}\,(a, tr'') = d$

$\Rightarrow$ {from (25)}

$\text{last write}\,(a, tr'') = d \,.$

8 Component refinement

The formal design of a caching node was presented as a single sequential process, maintaining three separate data objects. An alternative design consists of three separate processes, evolving concurrently, and cooperating to provide the same service. In this section, we present such a design, and show that it is a refinement of the sequential version.

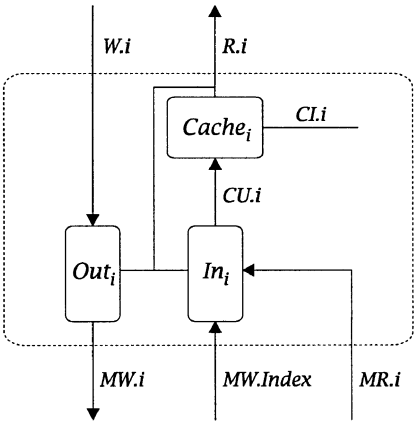

Fig. 3. Component refinement

The three components of our distributed design are an input queue, an output queue, and a cache. The input queue accepts memory reads and memory writes from the system, and issues cache updates. The output queue accepts write instructions from the processor, and issues memory writes to the rest of the memory system. The cache accepts updates, and provides data values.

All three processes are involved in events of the form $R.i.a.d$. The processor may read the contents of a memory address only when the cache is ready to provide the appropriate value *and* the input queue contains no locally-generated updates *and* the output queue is empty. At node i, the alphabets of the three processes are given by:

$$\begin{aligned}
&\alpha In_i \\
&\quad \mathrel{\widehat{=}} MW.Index.Address.Data \cup MR.i.Address.Data \cup \\
&\qquad CU.i.Index.Address.Data \cup R.i.Address.Data \ , \\
&\alpha Out_i \\
&\quad \mathrel{\widehat{=}} W.i.Address.Data \cup MW.i.Address.Data \cup \\
&\qquad R.i.Address.Data \ , \\
&\alpha Cache_i \\
&\quad \mathrel{\widehat{=}} CU.i.Index.Address.Data \cup R.i.Address.Data \cup \\
&\qquad CI.i.Address \ .
\end{aligned}$$

The in queue and the cache synchronize upon cache updates. The connectivity of the parallel combination is shown in Fig. 3.

The input queue In_i maintains a single state component in representing the sequence of updates pending. This process will: allow any read event if in contains no locally-generated updates; allow a cache update event using head in whenever in is non-empty; allow any memory write event, adding the received value to the end of the queue; allow any memory read event, adding a 0-tagged value to the end of the queue. This behaviour is modelled by a recursive choice process:

$$
\begin{aligned}
In_i\,(in) \mathrel{\widehat{=}} {} & \textsf{if no-local}_i\,(in)\ \textsf{then}\ R.i?a?d \rightarrow In_i\,(in)\\
& \Box\\
& \textsf{if}\ in \neq \langle\rangle\ \textsf{then}\ CU.i.j.a.d \rightarrow In_i\,(\textsf{tail}\ in)\\
& \qquad \textsf{where}\ (j,a,d) = \textsf{head}\ in\\
& \Box\\
& \Box_{j \in Index}\, MW.j?a?d \rightarrow In_i\,(in^\frown\langle(j,a,d)\rangle)\\
& \Box\\
& MR.i?a?d \rightarrow In_i\,(in^\frown\langle(0,a,d)\rangle)\ .
\end{aligned}
$$

The output queue allows read events only when it is empty. It will accept write requests from its processor, and add the appropriate pair to its queue. If the queue is non-empty, then a memory write may be performed using the head of the queue.

$$
\begin{aligned}
Out_i\,(out) \mathrel{\widehat{=}} {} & \textsf{if}\ out = \langle\rangle\ \textsf{then}\ R.i?a?d \rightarrow Out_i\,(out)\\
& \Box\\
& W.i?a?d \rightarrow Out_i\,(out^\frown\langle(a,d)\rangle)\\
& \Box\\
& \textsf{if}\ out \neq \langle\rangle\ \textsf{then}\ MW.i!a.d \rightarrow Out_i\,(\textsf{tail}\ out)\\
& \qquad \textsf{where}\ (a,d) = \textsf{head}\ out\ .
\end{aligned}
$$

The cache process will also allow the processor to read the contents of any address a, insisting that the current value $cache(a)$ is passed. It is ready to accept updates from the input queue. It may invalidate any address from its cache:

$$
\begin{aligned}
Cache_i(cache) \mathrel{\widehat{=}} {} & R.i?a!cache(a) \rightarrow Cache_i(cache)\\
& \Box\\
& CU.i?j?a?d \rightarrow Cache_i(cache \oplus \{a \mapsto d\})\\
& \Box\\
& \Box_{a \in \mathrm{dom} cache} CI.i.a \rightarrow\\
& \quad Node(in, out, \{a\} \lhd cache).
\end{aligned}
$$

We claim that the parallel combination of these three processes is a refinement of the sequential $Node_i$ process. Indeed, we can show that the two formulations are equivalent:

$$
\begin{aligned}
& Node_i(in, out, cache)\\
& \quad = In_i(in) \parallel Out_i(out) \parallel Cache_i(cache).
\end{aligned}
$$

Using the expansion law given in Sect. 2, we observe that the process

$$In_i\,(in) \parallel Out_i\,(out) \parallel Cache_i\,(cache)$$

is equivalent to the choice

$$
\begin{aligned}
& \textsf{if}\ out = \langle\rangle \wedge \textsf{no-local}_i\,(in)\ \textsf{then}\\
& \qquad R.i?a!cache(a) \rightarrow\\
& \quad (In_i\,(in) \parallel Out_i\,(out) \parallel Cache_i\,(cache))\\
& \Box\\
& MR.i?a?d \rightarrow\\
& \quad (In_i\,(in^\frown\langle(0,a,d)\rangle) \parallel Out_i\,(out) \parallel Cache_i\,(cache))\\
& \Box\\
& W.i?a?d \rightarrow\\
& \quad (In_i\,(in) \parallel Out_i\,(out^\frown\langle(a,d)\rangle) \parallel Cache_i\,(cache))\\
& \Box\\
& \textsf{if}\ in \neq \langle\rangle\ \textsf{then}\\
& \qquad CU.i.j.a.d \rightarrow\\
& \qquad (In_i(tl(in)) \parallel Out_i(out) \parallel Cache_i(cache \oplus \{a \mapsto d\}))\\
& \qquad\quad \textsf{where}\ (j,a,d) = \textsf{head}\ in\\
& \Box\\
& \textsf{if}\ out \neq \langle\rangle\ \textsf{then}\\
& \qquad MW.i!a.d \rightarrow\\
& \qquad (In_i(in^\frown\langle(i,a,d)\rangle) \parallel Out_i(\textsf{tail}out) \parallel Cache_i(cache))\\
& \qquad\quad \textsf{where}\ (a,d) = \textsf{head}\ out\\
& \Box\\
& \Box_{j \neq i} MW.j?a?d \rightarrow\\
& \quad (In_i\,(in^\frown\langle(j,a,d)\rangle) \parallel Out_i\,(out) \parallel Cache_i\,(cache))\\
& \Box\\
& \Box_{a \in \mathrm{dom}\, cache} CI.i.a \rightarrow\\
& \quad (In_i(in) \parallel Out_i\,(out) \parallel Cache_i(\{a\} \lhd cache).
\end{aligned}
$$

This tells us that the parallel combination satisfies the defining equation for the process $Node_i(in, out, cache)$. We may then appeal to the result stated in Sect. 2: that there is a unique solution to any collection of mutually-recursive equations, provided that the equations are properly guarded. These equations are clearly guarded – every recursive call is prefixed by an event – so we may conclude that the two designs are equivalent.

9 Discussion

In this paper we have shown that the language of CSP can be used to describe a lazy caching algorithm for shared memory. We also showed that the language of traces can be used to describe sequential consistency in a particularly concise fashion. We were then able to demonstrate that the protocol design is enough to guarantee sequential consistency of memory access.

The compositional nature of CSP meant that we could factor the verification into a number of smaller tasks. Following the structure of the design, we began by showing that each caching node satisfied a local consistency condition. We then demonstrated that the combination of these local conditions implied sequential consistency for the protocol as a whole.

Both the formal design and the verification were made easier by the fact that CSP permits multi-way synchronisation. We were able to represent a shared memory write as a single event, observable by all caching nodes. Furthermore, in our concurrent refinement of the components, we could express the requirement that the queue processes may block a read from the cache, by including the read events in their alphabets.

Appendix A: Proof of a component property

We may use the traces model to establish part of the claim – made in Sect. 6 – that each caching node meets a simple trace specification. We show

$Node_i\,(in, out, cache)$ **sat** $S(cache, tr)$,

where

$$S(cache, tr) \mathrel{\widehat{=}} \forall\, tr_0 \bullet tr_0 \frown \langle R.i.a.d\rangle \text{ prefix } tr \Rightarrow \text{last update}_i(a, tr_0, cache) = d.$$

The recursion induction theorem [Ros82] tells us that in order to show

$Node_i\,(in, out, cache)$ **sat** $S(cache, tr)$,

it is enough to prove this result under the assumption that all recursive calls to $Node_i\,(in', out', cache')$ already satisfy $S(cache', tr)$.

Let $tr \in traces\,[\![Node_i\,(in, out, cache)]\!]$. Then by the definition of $Node_i$:

$$\begin{array}{l}
tr = \langle\rangle \\
\vee \\
\exists\, tr', a', d', j \bullet \\
\quad tr = \langle R.i.a'.cache(a')\rangle \frown tr' \\
\qquad \wedge\, tr' \in traces\,[\![Node_i\,(in, out, cache)]\!] \\
\quad \vee\, tr = \langle MR.i.a'.d'\rangle \frown tr' \\
\qquad \wedge\, tr' \in traces\,[\![Node_i\,(in \frown \langle(0, a, d)\rangle, out, cache)]\!] \\
\quad \vee\, tr = \langle W.i.a'.d'\rangle \frown tr' \\
\qquad \wedge\, tr' \in traces\,[\![Node_i\,(in, out \frown \langle(a', d')\rangle, cache)]\!] \\
\quad \vee\, tr = \langle CU.i.j.a'.d'\rangle \frown tr' \\
\qquad \wedge\, tr' \in traces\,[\![Node_i\,(tl(in), out, \\
\qquad\qquad cache \oplus \{a' \mapsto d'\})]\!] \\
\quad \vee\, tr = \langle MW.i.a'.d'\rangle \frown tr' \\
\qquad \wedge\, tr' \in traces\,[\![Node_i\,(in \frown \langle(i, a', d')\rangle, \\
\qquad\qquad tl(out), cache)]\!] \\
\quad \vee\, tr = \langle MW.j.a'.d'\rangle \frown tr' \wedge i \neq j \\
\qquad \wedge\, tr' \in traces\,[\![Node_i(in \frown \langle(j, a', d')\rangle, out, cache)]\!] \\
\quad \vee\, tr = \langle CI.i.a'\rangle \frown tr' \\
\qquad \wedge\, tr' \in traces\,[\![Node_i(in, out, \{a'\} \lhd cache)]\!].
\end{array}$$

Hence, using the assumption about recursive calls, and rearranging:

$$\begin{array}{l}
tr = \langle\rangle \\
\vee \\
\exists\, tr', a', d', j \bullet \\
\quad \begin{pmatrix} tr = \langle R.i.a'.cache(a')\rangle \frown tr' \;\vee\; tr = \langle MR.i.a'.d\rangle \frown tr' \\ \vee\, tr = \langle W.i.a'.d'\rangle \frown tr' \;\vee\; tr = \langle MW.j.a'.d'\rangle \frown tr' \end{pmatrix} \\
\qquad \wedge\, S(cache, tr') \\
\quad \vee\, tr = \langle CU.i.j.a'.d'\rangle \frown tr' \wedge S(cache \oplus \{a' \mapsto d'\}, tr') \\
\quad \vee\, tr = \langle CI.i.a'\rangle \frown tr' \wedge S(\{a'\} \lhd cache, tr').
\end{array}$$

So if $tr_0 \frown \langle R.i.a.d\rangle$ prefix tr, then:

$$\begin{array}{l}
tr_0 = \langle\rangle \wedge\, d = cache(a) \\
\vee \\
tr_0 \neq \langle\rangle \\
\wedge\, hd(tr_0) \notin (CU.Index.Index.Address.Data \\
\qquad\qquad \cup CI.Index.Address) \\
\wedge \exists\, tr' \bullet tl(tr_0) \frown \langle R.i.a.d\rangle \text{ prefix } tr' \wedge S(cache, tr') \\
\vee \\
tr_0 \neq \langle\rangle \\
\wedge \exists\, tr', j, a', d' \bullet tl(tr_0) \frown \langle R.i.a.d\rangle \text{ prefix } tr' \\
\quad \wedge \begin{pmatrix} hd(tr_0) = CU.i.j.a'.d' \wedge S(cache \oplus \{a' \mapsto d'\}, tr') \\ \vee\, hd(tr_0) = CI.i.a' \wedge\; S(\{a'\} \lhd cache, tr') \end{pmatrix}.
\end{array}$$

So by the definition of last update$_i$ for the empty trace, and the definition of S:

$$\begin{array}{l}
tr_0 = \langle\rangle \wedge\, d = cache(a) = \text{last update}_i(a, tr_0, cache) \\
\vee \\
tr_0 \neq \langle\rangle \wedge\, hd(tr_0) \notin (CU.Index.Index.Address.Data \\
\qquad\qquad \cup CI.Index.Address) \\
\wedge\, d = \text{last update}_i(a, tl(tr_0), cache) \\
\vee \\
tr_0 \neq \langle\rangle \\
\wedge \exists\, tr', j, a', d' \bullet \\
\quad hd(tr_0) = CU.i.j.a'.d' \wedge \\
\qquad d = \text{last update}_i(a, tl(tr_0), cache \oplus \{a' \mapsto d'\}) \\
\quad \vee\, hd(tr_0) = CI.i.a' \wedge d \\
\qquad = \text{last update}_i(a, tl(tr_0), \{a'\} \lhd cache).
\end{array}$$

Hence from the definition of last update$_i$, performing a case analysis on a:

d = last update$_i$ $(a, tr_0, cache)$

as required.

Appendix B: Trace transformations

The proof of local sequential consistency requires two transformations upon traces. In this section, we exhibit two functions that will implement these transformations; they are presented in the style of a modern functional programming language.

Given trace tr of node i, we produce a locally-serial trace tr' such that tr and tr' are locally-consistent. This production may be expressed as the composition of two functions: $transform_1$ and $transform_2$. The first of these permutes the trace by moving the external read events; the second moves the external write events.

The required properties of these functions (Properties 6–10 in Section 6.2) may be proved inductively.

Read events

The function $transform_1$ moves all $R.i$ events forward (towards the start of the trace) to a position just after the last MW or $MR.i$ event whose corresponding $CU.i$ event occurs before the original position of the $R.i$; this is the last MW or $MR.i$ event that has had an effect on the cache before the read. If there are several consecutive reads, without any intervening cache updates, then these read events keep their same relative order.

$transform_1\ tr = transform_1'\ \langle\rangle\ tr$.

The function involves a single application of the recursive function $transform'_1$, which proceeds through the trace building a new trace as its first argument:

$$
\begin{aligned}
&transform'_1\ res\ \langle\rangle = res\\
&transform'_1\ res\ (\langle e\rangle^\frown tr) =\\
&\quad transform'_1\ (insert\ (\#(res \upharpoonright CU.i))e\ res)\ tr,\\
&\qquad\qquad \text{if } \exists\, a, d \bullet e = R.i.a.d\\
&\quad transform'_1\ (res^\frown\langle e\rangle)\ tr,\\
&\qquad\qquad \text{otherwise}.
\end{aligned}
$$

The subsidiary function $insert$ is defined such that $insert\ n\ r\ tr$ inserts the event r after the nth MW event and any adjacent read events:

$$
\begin{aligned}
&insert\ 0\ r\ \langle\rangle = \langle r\rangle\\
&insert\ 0\ r\ (\langle e\rangle^\frown tr) =\\
&\quad \langle e\rangle^\frown insert\ 0\ r\ tr, \quad \text{if } \exists\, a, d \bullet e = R.i.a.d\\
&\quad\ \langle r, e\rangle^\frown tr, \qquad\quad \text{otherwise}\\
&insert\ (n+1)\ r\ (\langle e\rangle^\frown tr) =\\
&\quad \langle e\rangle^\frown insert\ n\ r\ tr, \quad \text{if } \exists\, j, a, d \bullet e = MW.j.a.d \vee\\
&\qquad\qquad\qquad\qquad \exists\, a, d \bullet e = MR.i.a.d\\
&\quad\ \langle e\rangle^\frown insert\ (n+1)\ r\ tr,\\
&\qquad\qquad\qquad \text{otherwise}.
\end{aligned}
$$

Write events

The function $transform_2$ moves all $W.i$ events backwards (towards the end of the trace) to a position just before the corresponding $MW.i$ event; if there is no corresponding $MW.i$ event then the $W.i$ is moved to the end of the trace, but with all such events keeping their same relative order.

$$transform_2\ tr = transform'_2\ \langle\rangle\ tr\ .$$

The function involves an application of the recursive function $transform'_2$. The first argument of this function is used to store the sequence of write events still to be placed. The final application of the function yields the sequence of write events left at the end of the trace: these are the events that have yet to result in a matching memory write (MW event).

$$
\begin{aligned}
&transform'_2\ ws\ \langle\rangle = ws\\
&transform'_2\ ws\ (\langle e\rangle^\frown tr) =\\
&\quad transform'_2\ (ws^\frown\langle e\rangle)\ tr, \quad \text{if } \exists\, a, d \bullet e = W.i.a.d\\
&\quad\ \langle W.i.a.d, e\rangle^\frown transform'_2\ (ws - W.i.a.d)\ tr,\\
&\qquad\qquad\qquad\qquad \text{if } e = MW.i.a.d\\
&\quad\ \langle e\rangle^\frown transform'_2\ ws\ tr, \qquad \text{otherwise}.
\end{aligned}
$$

In the above definition, the expression $ws - W.i.a.d$ represents ws with the first occurence of $W.i.a.d$ removed.

Acknowledgements. We are grateful to Rob Gerth and to our other colleagues on the REACT project for many fruitful discussions. We are also grateful to the anonymous referees for their detailed comments upon an earlier version of this paper. This work was supported in part by ESPRIT Project 6021 “Building Correct Reactive Systems (REACT)” and by DRA Malvern.

References

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

[Bro83] Brookes SD: A Model for CSP. D. Phil thesis, Oxford University, 1983

[Dav93] Davies J: Specification and Proof in Real-Time Systems. Cambridge University Press, 1993, D. Phil thesis, Oxford University, 1991

[Dij76] Dijkstra EW: A Discipline of Programming. Englewood Cliffs, NJ: Prentice Hall 1976

[Ger95] Gerth R: Verifying sequentially consistent memory. Distrib Comput 12: 57–59 (1999)

[Hoa85] Hoare CAR: Communicating Sequential Processes. Englewood Cliffs, NJ: Prentice Hall 1985

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

[Low93] Lowe G: Probabilities and Priorities in Timed CSP. D. Phil thesis, Oxford University, 1993

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

[Ros82] Roscoe AW: A Mathematical Theory of Communicating Processes. D. Phil thesis, Oxford, 1982

[Spi87] Spivey JM: The Z Notation. Englewood Cliffs, NJ: Prentice Hall 1987

Gavin Lowe is a Lecturer in Computer Science at the University of Leicester. His research interests are in developing models of concurrency, especially based upon CSP, and using such models to reason about distributed systems. Recently, he has been particularly interested in applying CSP to the analysis of security protocols.

Jim Davies is a Lecturer in Computation at the University of Oxford, and a fellow of Kellogg College. He teaches on the University’s Software Engineering Programme. He studied mathematics and computation at New College, Oxford – his doctoral thesis was published in 1993 – and taught at the Universities of London and Reading before returning to Oxford in 1995. His research interests include the introduction of mathematical techniques into industrial software engineering, and the integration of information technology into computer science education.