Lazy Caching

YEHUDA AFEK

AT&T Bell Laboratories

GEOFFREY BROWN

Cornell University

AND MICHAEL MERRITT

AT&T Bell Laboratories

This paper examines cache consistency conditions for multiprocessor shared memory systems. It states and motivates a weaker condition than is normally implemented. An algorithm is presented that exploits the weaker condition to achieve greater concurrency. The algorithm is shown to satisfy the weak consistency condition. Other properties of the algorithm and possible extensions are discussed.

Categories and Subject Discriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories, shared memory*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors, multiple-instruction-stream, multiple-data-stream processors (MIMD)*; D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*

General Terms: Shared Memory

Additional Key Words and Phrases: Cache Coherence, Sequential Consistency

1 Introduction

The design of efficient shared memories is arguably the single most important problem in the design of parallel computer architectures. As with the specification of any component, the correctness condition satisfied by the shared memory constitutes a contract between

A preliminary version of this paper appeared in the *Proceedings of the 1989 ACM Symposium on Parallel Architectures and Algorithms*, (Sante Fe, New Mexico, June) ACM, New York, 1989, pp 209–222.

G. Brown's research was partially supported by National Science Foundation grant CCR-9058180 and a matching grant from AT&T.

Authors' present addresses:Y. Afek, Computer Science Department, Tel-Aviv University, Ramat-Aviv Israel 69978; G. Brown, School of Electrical Engineering, Phillips Hall, Cornell University, Ithaca NY 14853-5401; M. Merritt, 600 Mountain Ave., Murray Hill, NJ 07974

the architect and the programmer. In principle, the weaker the correctness condition, the more efficiently the memory can be implemented. At the same time, weaker conditions may complicate programming, particularly of operating systems and compilers. Being precise about shared-memory correctness conditions has profound implications for understanding both parallel architectures and their programming.

In this paper, we explore a formalization of Lamport's notion of sequential consistency [Lam79]. Typically, cache consistency algorithms implement much stronger conditions. For example, Lamport himself considers only algorithms satisfying stronger conditions than sequential consistency [Lam79]. Algorithms satisfying these conditions are behaviorally identical to a simple serial memory–a natural, but much stronger condition than sequential consistency. We describe an algorithm designed specifically to take advantage of as much concurrency as is practically possible, while remaining sequentially consistent. We note that logical anomalies may arise in sequentially consistent systems which have accurate measures of the passage of time, and discuss a trade-off between the observed accuracies of local clocks and queueing delays in implementations of our algorithm.

The problem of data consistency arises whenever systems allow multiple copies of a datum to exist; updating one of the copies results in an inconsistent system state which, if not properly handled, may result in erroneous system behavior. In multiprocessor systems the task of ensuring that cached copies of data are consistent falls upon the, so-called, consistency protocol. Traditional consistency protocols have required that all accessible copies of a datum be identical at all times [Smi82, SS86]. While a system implementing such a protocol is certainly correct, the discipline required to enforce such a property may unnecessarily constrain concurrency in the system.

Our formulation of sequential consistency is based on the intuition that a cache-based memory system should be indistinguishable, from the perspective of the processors, from a conventional shared memory system. Natural interpretations of "indistinguishable" give rise to successively weaker notions of correctness. For many settings, the condition we adopt is sufficiently strong to ensure that each behavior of a multi-process program executing on a cache-based system is a possible behavior of the same program executing on a conventional shared memory system.

We present a cache consistency protocol, the "lazy cache algorithm", that satisfies the weak sequential consistency condition. This algorithm is distinctly non-conventional in that it allows cache updates to be postponed at a cache while allowing the associated processor to continue to access the "stale" data held in the cache. Furthermore, we allow a processor to perform a sequence of writes to the memory system without requiring that each write be "completed" before the next write is performed. Such a system, illustrated in Figure 1, consists of a single shared memory and n processors each with an associated cache. Each of the caches has an output queue in which writes are buffered and an input queue in which update requests are buffered. Clearly, in order to satisfy any reasonable definition of

Figure 1: The Architecture of the Lazy Cache Algorithm

consistency, some restrictions must be placed upon accesses to a cache when either writes or updates are buffered; the lazy cache algorithm attempts to minimize these restrictions.

1.1 Previous Work

As noted above, the consistency definition explored in this paper is a formalization of *sequential consistency* originally discussed by Lamport [Lam79]: "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." However, Lamport goes on to consider only systems satisfying two conditions, the second of which is too restrictive: "Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue." Our cache algorithm does not satisfy this requirement, but it is sequentially consistent.

Similarly, much of the work on cache algorithms focuses on *cache coherence*, a stronger property than sequential consistency. "A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address" [CF78]. As noted by Scheurich and Dubois, this definition makes implicit architectural assumptions (specifically as to the atomicity of write operations) that limit its applicability to some multiprocessor architectures [SD87]. For systems in which this definition has a natural interpretation, coherence is usually the correctness condition cited. (See, e.g. [Smi82, KEW$^+$85, YYF85, SS86, Goo87].) As we understand the somewhat informal notion, memory systems that are cache coherent are behaviorally indistinguishable from serial memories, a stronger property than sequential consistency. (This distinction is made precise in Section 3, below.)

The idea of using less restrictive consistency conditions has been explored by other authors. Dubois, Scheurich, and Briggs consider algorithms in which write operations are not atomic, due to buffering, and also study the performance benefits of buffering memory accesses [DSB86]. Although they do not discuss buffering write operations at the writing processor (as the lazy cache algorithm does), their definition of *strong ordering* captures a broad class of sequentially consistent algorithms, including the lazy cache algorithm. While applicable to a broader class of architectures, like cache coherence the definition of strong ordering is somewhat informal and makes implicit architectural assumptions. * The conditions we discuss in Section 3 are stated as restrictions on the allowed sequences of events at an interface between the processors and a shared memory, and do not refer to any of the details of the memory implementation.

In a later paper, Scheurich and Dubois [SD87] present a sufficient condition for sequential consistency that is stronger than strong ordering, and show that it is satisfied by many conventional consistency algorithms. As they note, their consistency condition is a restriction of the general definition, and indeed, it excludes our algorithm's behavior. For example, one clause of their condition is the following: "If a processor issues a STORE then the processor may not issue another memory access until the value written has become *accessible* by *all other processors*." Our cache algorithm allows a processor to perform multiple writes (STOREs) without satisfying this condition.

The lazy cache algorithm is a generalization of an algorithm presented previously by Brown [Bro90]. The earlier algorithm buffered invalidation requests (our algorithm buffers cache updates) but not writes. Dually, the definition of consistency explored here is less restrictive than that presented by Brown. The definition used by Brown is based upon a notion of consistent states. Specifically, the system state following an action of one of the processors is consistent if the cache associated with that processor contains a subset of the data in shared storage in that state. An execution of the system is consistent if every state is consistent and the system is consistent if every possible execution is "equivalent" to a consistent execution. Thus, Brown's definition assumes a particular memory architecture consisting of a single shared memory and a cache associated with each processor. Our definition of consistency does not refer to system states but to the actions visible at the interface between the processors and the memory system. Thus, our formalization of sequential consistency makes no architectural assumptions.

Interesting recent work by Shasha and Snir [SS88] explores a similar weak consistency condition in the off-line context of compiling for shared memory multiprocessors. Conflict

*Recently, Adve and Hill have shown that strongly ordered memories are not in general sequentially consistent [AH90a].

relations are extracted from the complete program text, and minimal synchronization conditions are found. This work complements the on-line approach taken in the present paper.

The remainder of this paper is organized as follows. Section 2 presents a formal model for reasoning about multiprocessor memory systems. The consistency problem is discussed in section 3, where we define a consistent memory system. Section 4 presents the lazy cache algorithm and, in section 5, this algorithm is proven correct. Liveness and timing properties are addressed in section 6. The paper concludes with a discussion and additional problems.

2 Memory Systems

Following [LT87, Lyn88], a system is defined as a set of states and a set of actions which are multivalued functions from states to states. A system gives rise to a set of *executions*, which are finite or infinite sequences $s_0, \pi_1, s_1, ...$ of alternating states (the s_i's) and actions (the π_i's), where $s_i \in \pi_i(s_i - 1)$ and s_0 is an initial state. The actions of a system are partitioned into two classes: *external* actions and *internal* actions. The *behaviors* of a system are the sequences of external actions described by its executions; that is, the set of projections of the executions on the set of external actions. (If β is a behavior, and P is a set of actions, we denote by $\beta|P$ the subsequence of β consisting of the actions in P.)

An action is specified using the notation

$$\langle guard \rangle \;\rightarrow\; \langle command \rangle$$

where the *guard* is a boolean expression and the *command* is sequence of assignments. An action can only be executed from a state in which its guard is true. The guard of an action may be omitted if it is true in all states.

We are interested in the behavior of memory systems as viewed at the processor-memory interface. An n-processor shared memory is characterized by a particular set of external actions by which it interacts with the processors. In particular, such memory systems have four types of external actions:

$\mathsf{ReadRequest}_i(a)$ (processor i requests the data value in address a),

$\mathsf{ReadReturn}_i(d, a)$ (data d from address a returned to processor i),

$\mathsf{WriteRequest}_i(d, a)$ (processor i requests that data d be written to address a),

$\mathsf{WriteReturn}_i(d, a)$ (processor i is notified that data d has been written to address a).

We call these the *external memory actions*. Particular memory systems may have additional internal actions.

We define a conventional memory system, denoted M_{serial}, by specifying its actions and state variables.

In addition to the external actions above, M_{serial} has two internal actions, $\mathsf{Read}(d, a)$ and $\mathsf{Write}(d, a)$. These actions are the atomic events when the read and write operations actually occur.

The states of the conventional shared memory system, consist of two arrays:

- $Mem[address]$, with entries of type $Data$, which records the current state of the memory, and
- $Handshake[1...n]$, which takes as values either external actions of M_{serial} or $null$.

The array $Handshake$ stores outstanding memory operations: $Handshake[i] = \mathsf{WriteRequest}(d, a)$ indicates an outstanding write request by processor i of value d to memory location a, and $Handshake[i] = \mathsf{WriteReturn}(d, a)$ indicates the write operation has occurred and enables the response to processor i. Similarly, $Handshake[i] = \mathsf{ReadRequest}(a)$ indicates an outstanding read request by processor i of memory location a, and $Handshake[i] = \mathsf{ReadReturn}(d, a)$ indicates the read operation has obtained the value d and enables the response to processor i. Finally, $Handshake[i] = null$ indicates no outstanding request by processor i.

The start state of M_{serial} has all $Handshake$ entries $null$ and the array $Memory$ initialized to d_{init}.

The actions of M_{serial} are defined as follows.

$$
\begin{array}{lll}
\mathsf{ReadRequest}_i(a) :: & & Handshake[i] := \mathsf{ReadRequest}_i(a) \\[1em]
\mathsf{Read}_i(d,a) :: & & \\
Mem[a] = d & \rightarrow & Handshake[i] := \mathsf{ReadReturn}_i(d,a) \\
\ \wedge Handshake[i] = \mathsf{ReadRequest}_i(a) & & \\[1em]
\mathsf{ReadReturn}_i(d,a) :: & & \\
Handshake[i] = \mathsf{ReadReturn}_i(d,a) & \rightarrow & Handshake[i] := null \\[1em]
\mathsf{WriteRequest}_i(d,a) :: & & Handshake[i] := \mathsf{WriteRequest}_i(d,a) \\[1em]
\mathsf{Write}_i(d,a) :: & & \\
Handshake[i] = \mathsf{WriteRequest}_i(d,a) & \rightarrow & Mem[a] := d \\
 & & Handshake[i] := \mathsf{WriteReturn}_i(d,a) \\[1em]
\mathsf{WriteReturn}_i(d,a) :: & & \\
Handshake[i] = \mathsf{WriteReturn}_i(d,a) & \rightarrow & Handshake[i] := null
\end{array}
$$

$$i \in \{1, ..., n\},\ d \in data,\ a \in address$$

Thus, $\mathsf{ReadRequest}_i(a)$ can be executed from any state and has the effect of recording a request to read from address a, $\mathsf{Read}_i(d,a)$ can only be executed in a state where $Mem[a] = d$ and where processor i has requested data from address a, and $\mathsf{ReadReturn}_i(d,a)$ can only be executed in a state where $Handshake[i] = \mathsf{ReadReturn}_i(d,a)$, indicating that a read operation for i has occurred. Similarly, $\mathsf{WriteRequest}_i(d,a)$ can be executed from any state and has the effect of recording a write request, $\mathsf{WriteReturn}_i(d,a)$ can only be executed from a state in which $Handshake[i] = \mathsf{WriteRequest}_i(d,a)$ and has the effect of modifying Mem by setting the value of $Mem[a]$ to d and of setting $Handshake[i]$ to $\mathsf{WriteReturn}_i(d,a)$, and $\mathsf{WriteReturn}_i(d,a)$ can only be executed in a state where $Handshake[i] = \mathsf{WriteReturn}_i(d,a)$, indicating that a write of d to a for i has occurred. †

† The external actions of M_{shared} model a handshake between each processor and the shared memory; the $\mathsf{ReadRequest}_i$ and $\mathsf{WriteRequest}_i$ actions model inputs from processor i to the memory, and the $\mathsf{ReadReturn}_i$ and $\mathsf{WriteReturn}_i$ actions are outputs from the memory to processor i. If a processor violates the handshake (by issuing a new request before a reply has been output) the previous request is overwritten. If the previous request was for a write operation, the write may or may not have occurred, depending on whether the enabled internal Write action had fired. A less constraining specification might permit other behavior in the presence of such an erroneous environment.

3 Consistency Conditions for Shared Memories

Consistency conditions for shared memories are generally based on the intuition that memories should look to the processors like M_{serial}. There are several successively weaker definitions one might choose that fit this intuition.

To describe these conditions, it is notationally convenient to identify the subsets of external actions with respect to a single processor. Thus, P_i denotes the set of actions $\{\mathsf{ReadRequest}_i(a), \mathsf{ReadReturn}_i(d,a), \mathsf{WriteRequest}_i(d,a), \mathsf{WriteReturn}_i(d,a) : d \in data,\ a \in address\}$. Hence, $\beta|P_i$ denotes the sequence formed from β by deleting all actions except the external actions of processor i.

Definition 1 (Coherence) *A memory M is consistent only if, for each of its behaviors β, $\beta|\bigcup_i P_i$ is a behavior of M_{serial}.*

Definition 2 (Sequential Consistency) *A memory M is consistent only if, for each of its behaviors β, there exists a behavior β_s of M_{serial} such that for all processors i, $\beta|P_i = \beta_s|P_i$.*

Definition 3 *A memory M is consistent only if, for each of its behaviors β and for each processor i, there exists a behavior β_{s_i} of M_{serial} such that $\beta|P_i = \beta_{s_i}|P_i$.*

Definition 1 is the strongest of the three: it requires that the behavior visible at the processor-memory interface of a cache memory system should correspond exactly to some behavior of the conventional shared memory system when viewed at the same interface. (E.g. to *all* the processors simultaneously.) Systems satisfying this condition are essentially implementations of M_{serial}. This definition is our characterization of *cache coherence* as a pure behavioral specification, abstracted from any architectural assumptions. This is the condition usually used in designing cache algorithms, and has also been called *dynamic atomicity* [HA90]. (See also [Mis86] for a axiomatic specification.)

Definition 2 is the consistency definition advocated in this paper. It is less restrictive than definition 1, requiring that for each behavior of the cache memory system, there exist a behavior of the conventional shared memory so that for each processor individually, the two behaviors are identical at the interface of that processor with the memory system. (This is similar to the serializability condition that most concurrency control algorithms impose on committed transactions.) Put another way, a shared memory is consistent if all of the processors' local views can be interleaved to form a single serial behavior, regardless of the actual relative ordering of events at different processors. Just as the first definition formalizes cache coherence, this second definition is our characterization of *serial consistency* as a pure behavioral specification.

Finally, definition 3 merely requires that for each behavior β of the cache memory system and each processor i, there exists a behavior β_{s_i} of the conventional shared memory such that the behaviors at the interface of processor i with the memory systems are identical. However, β_{s_i} could be different than β_{s_j} if $i \neq j$. (This is similar to the conditions that locking algorithms impose on all active transactions: Occasionally, transactions must be aborted because their individual histories cannot be merged into a single serial history satisfying definition 2.)

Certainly definition 1 is safe from the point of view of applications, but it may too severely restrict the efficiency of the possible implementations. On the other hand, definition 3 is almost certainly too unrestrictive. It is satisfied, for example, by an architecture in which each processor has its own independent local memory. We suggest that definition 2 is strong enough to be useful in many interesting systems, while permitting architectures that may be more efficiently implementable than those that satisfy definition 1.‡

In particular, definition 2 requires that there exist a single behavior of the M_{serial} that is consistent with the local views of each of the processors. At the same time, the order in which actions of different processors occur may be different in the actual behavior of the system than in the behavior that is being simulated. For example, the following sequence satisfies definition 2, but not definition 1, assuming $d \neq d_{init}$ where d_{init} is the initial value of $Mem[1]$.

$$\mathsf{WriteRequest}_1(d,1), \mathsf{Write}_1(d,1), \mathsf{WriteReturn}_1(d,1),$$
$$\mathsf{ReadRequest}_2(1), \mathsf{Read}_2(d_{init},1), \mathsf{ReadReturn}_2(d_{init},1)$$

Such examples are probably more easily understood when describe graphically, as below.

1 $\vdash\mathsf{Write}(d,1)\dashv$
2 $\qquad\qquad\qquad\vdash\mathsf{Read}(d_{init},1)\dashv$

Here, the rows correspond to events at the indicated processor, and time increases from left to right. The notation $\vdash\mathsf{Write}(d,1)\dashv$ in the first row denotes the sequence of events $\mathsf{WriteRequest}_1(d,1), \mathsf{Write}(d,1), \mathsf{WriteReturn}_1(d,1)$; similarly, $\vdash\mathsf{Read}(d_{init},1)\dashv$ in the second row denotes the handshake $\mathsf{ReadRequest}_2(1), \mathsf{Read}_2(d_{init},1), \mathsf{ReadReturn}_2(d_{init},1)$.

The corresponding behavior of M_{serial} is of course:

$$\mathsf{ReadRequest}_2(1), \mathsf{Read}_2(d_{init},1), \mathsf{ReadReturn}_2(d_{init},1)$$
$$\mathsf{WriteRequest}_1(d,1), \mathsf{Write}_1(d,1), \mathsf{WriteReturn}_1(d,1).$$

And, graphically:

‡While definition 3 is of dubious interest, recent work has explored more useful conditions that are less restrictive than sequential consistency (definition 2) See, for example, [AH90b, GLL+90, GMG91, HA90, LS88, ABHN91]. These approaches all involve some complication of the programming model, while sequential consistency allows the programmer to assume the memory is serial.

1		$\vdash\mathsf{Write}(d, 1)\dashv$
2	$\vdash\mathsf{Read}(d_{init}, 1)\dashv$	

Thus, the read and write operations of processors 1 and 2 happen in different orders in the simulated behavior than in the actual behavior.

Below is an example of a behavior that satisfies definition 3 but not 2, assuming that d_{init}, d_1 and d_2 are all distinct: (Henceforth, we present examples graphically and omit the sequences they represent.)

1	$\vdash\mathsf{Write}(d_1, 1)\dashv$					
2		$\vdash\mathsf{Write}(d_2, 1)\dashv$				
3			$\vdash\mathsf{Read}(d_1, 1)\dashv$	$\vdash\mathsf{Read}(d_2, 1)\dashv$		
4					$\vdash\mathsf{Read}(d_2, 1)\dashv$	$\vdash\mathsf{Read}(d_1, 1)\dashv$

The following behavior of M_{serial} preserves the order of actions of Processor 3.

1	$\vdash\mathsf{Write}(d_1, 1)\dashv$					
2				$\vdash\mathsf{Write}(d_2, 1)\dashv$		
3		$\vdash\mathsf{Read}(d_1, 1)\dashv$			$\vdash\mathsf{Read}(d_2, 1)\dashv$	
4			$\vdash\mathsf{Read}(d_1, 1)\dashv$			$\vdash\mathsf{Read}(d_2, 1)\dashv$

Meanwhile, the following behavior of M_{serial} preserves the order of actions of Processor 4.

1				$\vdash\mathsf{Write}(d_1, 1)\dashv$		
2	$\vdash\mathsf{Write}(d_2, 1)\dashv$					
3		$\vdash\mathsf{Read}(d_2, 1)\dashv$			$\vdash\mathsf{Read}(d_1, 1)\dashv$	
4			$\vdash\mathsf{Read}(d_2, 1)\dashv$			$\vdash\mathsf{Read}(d_1, 1)\dashv$

But there is no single behavior consistent with the conflicting order in which processors 3 and 4 see the two writes by processors 1 and 2.

Motivation for the suggested correctness condition (2, above) rests heavily on the presumed architecture of the system. If processors have an independent means of communication, or accurate measures of real time, it is possible for them to observe anomalies by comparing the relative times of different memory operations. We postpone discussion of these issues to Section 6, where we also describe extensions of the algorithm of the next section that address some of these synchronization issues.

The correctness conditions discussed above are all safety properties, constraining shared memories when they respond to requests. A discussion of liveness properties, which ensure that some response is actually made to requests, is also postponed to Section 6.

4 The Lazy Cache Algorithm

In this section we present an algorithm for implementing a concurrent cache memory system that satisfies our formal sequential consistency condition (definition 2). It is a generalization of typical cache-based memories, but allows greater flexibility in implementation by decreasing the constraints on global synchronization.

4.1 Overview of the Algorithm

In addition to the central memory Mem, a private cache C_i is associated with each processor i. Unlike typical algorithms, in the lazy cache algorithm these arrays may become "out of date", but still be referenced by their associated processor. In addition, the central memory array may itself become out of date, with respect to processors' views. The overall effect is that some reads appear to be moved back in time, as they read out of date cache values, and writes are moved forward in time, as their effects appear after the associated processor finishes the operation.

As indicated in Figure 1, this process is managed by a pair of queues, Out_i and In_i, associated with each cache. The Out_i queue buffers write operations by processor i, and the In_i queue buffers update operations to the cache C_i. Instead of updating the Mem array as in M_{serial}, action $\mathsf{WriteReturn}_i(d,a)$ by processor i appends the pair (d,a) to Out_i. As a separate atomic $\mathsf{MemoryWrite}_i$ action, the entry at the head of Out_i is appended to all In queues and stored in the array Mem. This effectively delays the apparent occurrence of the write until the appropriate entry reaches the head of Out_i. Occurrences of the $\mathsf{CacheUpdate}_i$ action use the entries in In_i to update the contents of the cache C_i.

Suppose that processor i writes value d to memory location a and reads the same location, with no intervening operations by other processors. The value returned must be the value written. Similarly, if one processor observes two writes to occur in a particular order, all processors must observe the same relative order. To assure that these constraints are satisfied, the algorithm ensures that the state of the cache C_i used by a read operation reflects values in Mem that are at least as current as the most recent write by the same processor. Accordingly, a read operation cannot follow a write by the same processor until the entry for that write moves out of Out_i and the associated update moves through In_i queue into C_i.

As Theorem 1 below demonstrates, this restriction is sufficient to guarantee that the finite behaviors of M_{cache} satisfy Definition 2. Corollary 11 argues that, under liveness assumptions prohibiting infinite queueing delays, this property extends to infinite behaviors of M_{cache}, as well.

To accomplish this, entries in the In queues that record writes by processor P_i are tagged with a special character ("*"). Responses to reads ($\mathsf{ReadReturn}'_i s$) are not allowed to proceed unless the Out_i queue is empty and there are no tagged entries in In_i.

Two types of internal actions are enabled in every state: $\mathsf{MemoryRead}_i$ actions add any $(data, address)$ pair from Mem to In_i, and $\mathsf{CacheInvalidate}_i$ actions remove entries from C_i. By allowing these actions nondeterministically from any state, we avoid specifying the memory-management strategy that might be chosen to enhance efficiency in any particular system. Our proof of correctness applies *a fortiori* to such strategies by restriction of nondeterminism.

Such an algorithm may be more efficiently implementable than more traditional algorithms because processors are only delayed during read and write operations in the case that a read follows a write or does not find the needed value in the cache. Furthermore, global synchronization occurs at the events appending entries to the In queues, not at the cache arrays C_i. In an implementation using concurrent queues, other processors could continue to access their local caches in parallel with the global atomic update of the In queues. This compares favorably, for example, with algorithms that lock all the processors out of their caches during invalidation activity.

Note that M_{cache} makes no effective progress unless some mechanism flushes the queues and enforces a reasonable memory-management strategy on the cache contents. Without specifying a particular mechanism, the discussion of liveness properties in Section 6 analyzes the behavior of M_{cache} under the assumption that queues are flushed and requested data values are brought into the cache.

4.2 A Precise Description of the Algorithm

The cache memory system M_{cache} has the same external actions as M_{serial}, but has more complex data structures and additional internal actions. The data structures include Mem of M_{serial} and the following:

- n pairs of unbounded fifo queues, or lists, In_i and Out_i, the entries in which are either $(data, address)$ or $(data, address, *)$ tuples. These queues have the following operations:
  - $append(queue, item)$, adds $item$ as the last entry in $queue$.
  - $head(queue)$, returns the first entry in $queue$.
  - $tail(queue)$, returns the result of removing $head(queue)$ from $queue$.
  - $\{\}$ denotes the empty queue.
  - $queue[i]$ denotes the ith element of $queue$ where $queue[0] = head(queue)$

- n partial functions $C_i : Address \times Data$. These functions may be updated via two operations:
  - $update(C_i, d, a)$ returns the partial function which maps address a to value d, but is otherwise identical to C_i.
  - $restrict(C_i)$, which nondeterministically chooses any partial function that is a subset of C_i.

The initial states of M_{cache} are those in which all queues are empty, the array Mem is initialized to d_{init}, and the $C_i \subseteq Mem, \forall i \in \{1, ..., n\}$.

In addition to the external actions of M_{serial}, the actions of M_{cache} include the internal actions discussed above: $\mathsf{MemoryWrite}_i(d, a)$, $\mathsf{MemoryRead}_i(d, a)$, $\mathsf{CacheUpdate}_i(d, a)$, $\mathsf{CacheInvalidate}_i$. The actions of M_{cache} are specified as follows.

$\mathsf{ReadRequest}_i(a)$::		$Handshake[i] := \mathsf{ReadRequest}_i(a)$
$\mathsf{ReadReturn}_i(d,a)$::		
$Handshake[i] = \mathsf{ReadRequest}_i(a)$ $\wedge C_i[a] = d$ $\wedge Out_i = \{\}$ $\wedge$ (There are no starred entries in In_i.)	$\rightarrow$	$Handshake[i] := null$
$\mathsf{WriteRequest}_i(d,a)$::		$Handshake[i] := \mathsf{WriteRequest}_i(d,a)$
$\mathsf{WriteReturn}_i(d,a)$::		
$Handshake[i] = \mathsf{WriteRequest}(d,a)$	$\rightarrow$	$Handshake[i] := null;$ $Out_i := append(Out_i, (d,a))$
$\mathsf{MemoryWrite}_i(d,a)$::		
$head(Out_i) = (d,a)$	$\rightarrow$	$Mem[a] := d;$ $Out_i := tail(Out_i);$ $(\forall k \neq i \| :: In_k := append(In_k, (d,a)))$ $(In_i := append(In_i, (d,a,*)))$
$\mathsf{MemoryRead}_i(d,a)$::		
$Mem[a] = d$	$\rightarrow$	$In_i := append(In_i, (d,a))$
$\mathsf{CacheUpdate}_i(d,a)$::		
$head(In_i)$ is either (d,a) or $(d,a,*)$.	$\rightarrow$	$In_i := tail(In_i);$ $C_i := update(C_i, d, a)$
$\mathsf{CacheInvalidate}_i$::		$C_i := restrict(C_i)$

$$i \in \{1, ..., n\},\ a \in addresses,\ d \in data$$

5 A Proof of Correctness of M_{cache}

5.1 Proof Strategy

To show that M_{cache} satisfies our correctness criterion for shared memories, we construct from every behavior β of M_{cache} a behavior β_s of M_{serial} such that $\beta|P_i = \beta_s|P_i, \forall i \in \{1, ..., n\}$.

To facilitate the construction of β_s, it is helpful to augment the states of M_{cache} with *history variables*. That is, from M_{cache} we construct a new system M^h_{cache}, which has additional state components that record useful information about ongoing executions. The construction of M^h_{cache} will be such that it is trivial to show that M_{cache} and M^h_{cache} have identical behaviors.

Using the additional state information, we can easily prove invariants about the states of M^h_{cache}. Given a behavior β of M_{cache}, we know it is a behavior also of M^h_{cache}, and so has a corresponding execution α of M^h_{cache}. From this execution α we construct a behavior β_s of M_{serial} with the required properties.

5.2 Memory System M^h_{cache}

The states of M^h_{cache} include the following components in addition to those in M_{cache}:

- $WriteCounter$, a variable of type integer, which counts the writes as they are applied to the array Mem, effectively associating a sequence number with them.
- $Hist_M$, a sequence of states of Mem.
- n variables $Lastin_i, \forall i \in \{1, ..., n\}$, of type integer, which record the sequence number of the last update applied to C_i.
- n variables $Lastout_i, \forall i \in \{1, ..., n\}$, of type integer, which record the sequence number of the last write by P_i applied to Mem.
- The write entries of the queues $In_1, ..., In_n$, which in M_{cache} are either $(data, address)$ or $(data, address, *)$ tuples, are extended in M^h_{cache} with a final integer field, which records the sequence number of the associated write operation. (Thus, the queue entries are either $(data, address, integer)$ or $(data, address, *, integer)$ tuples.)

Initially, $WriteCounter$, the $Lastin_i$ and $Lastout_i$ have value 0 and $Hist_M$ is the sequence containing the initial state of Mem.

The actions of M^h_{cache} are the same as for M_{cache}, with the following exceptions:

MemoryWrite$_i(d, a)$::
$head(Out_i) = (d, a) \quad \rightarrow \quad Mem[a] := d;$
$WriteCounter := WriteCounter + 1;$
$LastOut_i := WriteCounter;$
$Out_i := tail(Out_i);$
$(\forall k \neq i \| :: In_k := append(In_k, (d, a, WriteCounter)));$
$In_i := append(In_i, (d, a, *, WriteCounter));$
$Hist_M := append(Hist_M, Mem)$

MemoryRead$_i(d, a)$::
$Mem[a] = d \quad \rightarrow \quad In_i := append(In_i, (d, a, WriteCounter))$

CacheUpdate$_i(d, a)$::
$head(In_i)$ is either (d, a, wc) or $(d, a, *, wc)$, for some wc. $\rightarrow$ $In_i := tail(In_i);$
$C_i := update(C_i, d, a);$
$Lastin_i := wc$

$$i \in \{1, ..., n\},\ a \in addresses,\ d \in data,$$
$$wc \in integer$$

Proposition 1 *The systems M_{cache} and M^h_{cache} have identical sets of behaviors.*

5.3 Consistency of M^h_{cache}

Lemma 2 *The following invariants are maintained in all executions of M^h_{cache}, $(\forall i \in \{1, ..., n\})$.:*

1. *$Hist_M[WriteCounter] = Mem$; furthermore, $Hist_M$ is a sequence containing $WriteCounter + 1$ states of Mem.*
2. *$(Lastin_i \leq WriteCounter) \wedge (Lastout_i \leq WriteCounter)$.*
3. *The order of the sequence numbers along the entries of In_i form a non-decreasing sequence between $Lastin_i$ and $WriteCounter$.*
4. *$Out_i = \{\} \wedge$ (There are no starred entries in In_i) $\Rightarrow$ $Lastin_i \geq Lastout_i$.*
5. *Let wc be the sequence number of the tail entry in In_i, or $Lastin_i$ if In_i is empty. Then $wc = WriteCounter$.*
6. *Let p be any prefix of In_i, let wc be sequence number of the last entry in p, or $Lastin_i$ if p is empty, and let $update(C_i, p)$ be the partial function obtained by successively updating C_i with the $(data, address)$ pairs from entries in p. Then $update(C_i, p) \subset Hist_M[wc]$.*

7. $C_i \subset Hist_M[Lastin_i]$.

Proof: Proving the invariants requires checking that they are maintained by the nine actions of M^h_{cache}. For example, consider the action $\mathsf{MemoryWrite}_i$. This action appends the modified Mem to $Hist_M$ and increments $WriteCounter$ preserving (1); assigns $WriteCounter$ to $LastOut_i$ preserving (2); adds $(d, a, WriteCounter)$ to all In_k, $k \neq i$ and $(d, a, *, WriteCounter)$ to In_i,, preserving (3) and (5); and leaves the premise of (4) false; now $Hist_M[WriteCounter] = Mem$ and $Mem = update(Hist_M[WriteCounter - 1], d, a)$, preserving (6); and, preserves(7) since none of C_i, $Lastin_i$, or $Hist_M[Lastin_i]$ are modified. ∎

Our goal now is to construct a behavior β_s of M_{serial} from any behavior β of M^h_{cache}, such that $\beta_s|P_i = \beta|P_i, \forall i \in \{1, ..., n\}$. We do so by looking at an execution α of M^h_{cache} with behavior β, and extract an important ordering relationship between events in β. The sequences which are candidates for the simulated serial behavior β_s are shown to be those which extend this ordering relationship.

Given any state of an execution α of M^h_{cache}, each entry (d, a) in queue Out_i was originally inserted as the consequence of a particular $\mathsf{WriteReturn}_i(d, a)$ event. When an entry (d, a) in Out_i is dequeued (by a $\mathsf{MemoryWrite}_i(d, a)$ event), the $WriteCounter$ variable is incremented. Thus, every WriteReturn event in α is associated either with a unique integer from 1 to $WriteCounter$ or with a unique entry in one of the Out_i queues. Furthermore, every such integer and queue entry has its unique associated WriteReturn event.

Given any finite execution α of M^h_{cache}, we define a mapping WC^W_α from the WriteReturn events in α to the set $\{1, 2, ..., WriteCounter\} \cup \{q | q$ is an entry in Out_i for some $i\}$. The mapping is defined inductively, and is initially empty. As the execution continues, the particular correspondence between a $\mathsf{WriteReturn}_i$ event π and the associated Out_i queue entry $WC^W_\alpha(\pi)$ persists until $WC^W_\alpha(\pi)$ is dequeued. Following this step, π is associated with the new value of $WriteCounter$, and this correspondence is stable. A simple induction establishes the following.

Proposition 3 *Given any finite execution α of M^h_{cache}, WC^W_α is a one-to-one and onto mapping from the* WriteReturn *events in α to the set $\{1, 2, ..., WriteCounter\} \cup \{q | q$ is an entry in Out_i for some $i\}$.*

In addition, a second function WC^R_α maps the ReadReturn events to the integers $\{0, 1, ..., WriteCounter\}$, where $WC^R_\alpha(\mathsf{ReadReturn}_i)$ is the value of $Lastin_i$ at the time that $\mathsf{ReadReturn}_i$ occurred in α.

Finally, for a finite execution α of M^h_{cache}, we define a mapping WC_α from the set of WriteReturn and ReadReturn events in α to the set $\{0, 1, ...\} \cup \{\infty\}$, as follows. If π is a

ReadReturn event, $WC_\alpha(\pi) = WC^R_\alpha(\pi)$. If π is a WriteReturn event, and $WC^W_\alpha(\pi)$ is integer-valued, then $WC_\alpha(\pi) = WC^W_\alpha(\pi)$. But if π is a WriteReturn event and $WC^W_\alpha(\pi)$ is a queue entry, then $WC_\alpha(\pi) = \infty$.

The mapping WC_α illuminates the logic of the lazy cache algorithm. The sequence of integer values it associates with the WriteReturn events is precisely the order of the WriteReturns in the simulated serial behavior, and corresponds to the order in M^h_{cache} in which the write entries are removed from the Out_i queues and applied to Mem. Those WriteReturn events π for which $WC_\alpha(\pi)$ is ∞ represent write operations which have already returned to the calling processor, but have yet to be applied to Mem by MemoryWrite. (Hence, the order in which they will be applied to Mem remains undetermined.) For ReadReturn operations, the mapping WC_α determines which WriteReturn operation most immediately precedes the ReadReturn in the simulated serial behavior. Note that WC_α is only a partial order on WriteReturn and ReadReturn events. Clearly all π where $WC_\alpha(\pi) = \infty$ are only partially ordered with respect to other writes. In addition, each integer value of WC corresponds to a single WriteReturn and potentially several ReadReturn's.

The following technical results allow us to reason about event sequences in the case analysis of Theorem 1. They all follow by simple induction.

Proposition 4 *Let α be a finite execution of M^h_{cache}. If π is a* WriteReturn$_i(d,a)$ *event in execution α such that $WC_\alpha(\pi) = j \neq \infty$, then $Hist_M[j] = update(Hist_M[j-1], d, a)$ and $j > 0$. Furthermore,* WriteRequest$_i(d,a)$ *is the action of processor i immediately preceding π in α.*

Proposition 5 *Let α be a finite execution of M^h_{cache}. If π is a* ReadReturn$_i(d,a)$ *event where $WC_\alpha(\pi) = j$, then $Hist_M[j][a] = d$. Also,* ReadRequest$_i(a)$ *is the action of processor i immediately preceding π in α.*

Proposition 6 *Let α be a finite execution of M^h_{cache}. If π is an event in α such that $0 < WC_\alpha(\pi) < \infty$, then there is a (unique)* WriteReturn *event ϕ in α such that $WC_\alpha(\phi) = WC_\alpha(\pi)$. Furthermore, if $WC_\alpha(\pi) > 1$, then there exists an event η in α such that $WC_\alpha(\eta) + 1 = WC_\alpha(\pi)$.*

Definition *Let α be a finite execution of M^h_{cache}. Let $<_\alpha$ be the relation on events in α satisfying the following conditions, for all events π and ϕ in α.*

1. *If π and ϕ are actions of P_i for some i, and π precedes ϕ in α, then $\pi <_\alpha \phi$.*
2. *If π and ϕ are* WriteReturn *or* ReadReturn *events and $WC_\alpha(\pi) < WC_\alpha(\phi)$ then $\pi <_\alpha \phi$.*

3. *If* π *is a* WriteReturn *event,* ϕ *is a* ReadReturn *event and* $WC_\alpha(\pi) = WC_\alpha(\phi)$ *then* $\pi <_\alpha \phi$.

Lemma 7 *Let* α *be a finite execution of* M^h_{cache}*. Then the transitive closure of* $<_\alpha$ *is irreflexive.*

We postpone the proof of Lemma 7 to the appendix.

Theorem 1 *Let* α *be a finite execution of* M^h_{cache} *with behavior* β*. Let* β_s *be a sequence determined by any total ordering of the events in* β *that extends* $<_\alpha$*. Then* β_s *is a behavior of* M_{serial} *such that* $\beta_s|P_i = \beta|P_i, \forall i \in \{1, ..., n\}$*.*

Proof: By Lemma 7, a reordering β_s of the events in β that is consistent with $<_\alpha$ exists. The equality $\beta_s|P_i = \beta|P_i$ is immediate from the definition of the relation $<_\alpha$. It remains to show that β_s is a behavior of M_{serial}. We do this by constructing an execution α_s of M_{serial} from β_s. The construction of α_s is inductive on the prefixes β'_s of β_s. A characteristic of this construction is that if $WC_\alpha(\phi) < \infty$ for WriteReturn or ReadReturn action ϕ, then in the state following ϕ in α_s, $Mem = Hist_M[WC_\alpha(\phi)]$. The execution corresponding to the empty prefix is simply the initial state of M_{serial} (which has the value $Hist_M[0]$). Assume that α'_s is the finite execution derived for prefix β'_s of β_s, and that the final state of α'_s is r. If $\beta'_s\pi$ is a prefix of β_s, where π is a single action, then the corresponding execution is $\alpha'_s\gamma$, where γ is a sequence of (one or two) actions and intervening states, defined as follows: (We use the notation r^x_v to denote the state obtained from state r by assigning to state variable x the value v.)

- $\pi = \mathsf{ReadRequest}_i(a)$: $\gamma = \mathsf{ReadRequest}_i(a) r^{Handshake[i]}_{\mathsf{ReadRequest}_i(a)}$
- $\pi = \mathsf{ReadReturn}_i(d,a)$: $\gamma = \mathsf{Read}_i(d,a) r^{Handshake[i]}_{\mathsf{ReadReturn}_i(d,a)} \mathsf{ReadReturn}_i(d,a) r^{Handshake[i]}_{null}$
  It remains to show that action $\mathsf{ReadReturn}_i(d,a)$ is enabled in state r, i.e. $Mem[a] = d$ and $Handshake[i] = \mathsf{ReadRequest}_i(a)$, and that $Mem = Hist_M[WC_\alpha(\pi)]$ in state r. ¿From proposition 5 and the definition of $<_\alpha$, $\mathsf{ReadRequest}_i(a)$ is the action of processor i immediately preceding π in β'_s. ¿From proposition 6 and the definition of $<_\alpha$ either π is the first Return event in β_s, in which case $WC_\alpha(\pi) = 0$, or $WC_\alpha(\eta) = WC_\alpha(\pi)$ for the Return event η immediately preceding π in β_s. In the first case the requirements follow from the initial conditions and in the second, from invariant preserved by the construction of α_s.
- $\pi = \mathsf{WriteRequest}_i(d,a)$: $\gamma = \mathsf{WriteRequest}_i(d,a) r^{Handshake[i]}_{\mathsf{WriteRequest}_i(d,a)}$

- $\pi = \mathsf{WriteReturn}_i(d,a)$:
  $\gamma = \mathsf{Write}_i(d,a) r^{Mem,Handshake[i]}_{update(Mem,d,a),\mathsf{WriteReturn}_i(d,a)} \mathsf{WriteReturn}_i(d,a) r^{Mem,Handshake[i]}_{update(Mem,d,a),null}$

  ¿From proposition 4, $Handshake[i] = \mathsf{WriteRequest}_i(d,a)$ in state r, hence π is enabled. It remains to show that if $WC_\alpha(\pi)$ is defined then $Mem = Hist_M[WC_\alpha(\pi)]$ in state $r^{Mem,Handshake[i]}_{update(Mem,d,a),null}$. ¿From proposition 4 it suffices to show that $Mem = Hist_M[WC_\alpha(\pi)-1]$ in state r. From proposition 6 and the definition of $<_\alpha$, either π is the first Return action in β_s, in which case $WC_\alpha(\pi) = 1$, or $WC_\alpha(\eta)+1 = WC_\alpha(\pi)$ for the Return action η immediately preceding π in β_s; in either case, the condition holds.

∎

Corollary 8 *If α is a finite execution of M_{cache} with behavior β, then there exists a behavior β_s of M_{serial} such that $\beta_s|P_i = \beta|P_i$, $\forall i \in \{1,...,n\}$.*

Proof: Immediate from Proposition 1 and Theorem 1. ∎

to

6 Liveness and Other Properties

In this section we discuss and state four additional properties of the lazy cache algorithm. Three of the observations made here are liveness-related and the fourth one states that there are behaviors satisfying Definition 2 which cannot be produced by the lazy cache algorithm.

6.1 Liveness

We define the following *liveness constraints* on executions of M_{cache} and M^h_{cache}.

For all $1 \le i \le n$, $a \in address$ and $d \in data$:

L1 $In_i \neq \{\}$ leads to $\mathsf{CacheUpdate}_i$.

L2 $Out_i \neq \{\}$ leads to $\mathsf{MemoryWrite}_i$.

L3 $(Handshake[i] = \mathsf{ReadRequest}_i(a)) \wedge (C_i[a]$ is defined.) leads to $\mathsf{ReadReturn}_i$.

L4 $(Handshake[i] = \mathsf{ReadRequest}_i(a)) \wedge (C_i[a]$ is not defined.) $\wedge$ (No entry in In_i has address a.) leads to $\mathsf{MemoryRead}_i(d,a)$ for some d.

L5 $Handshake[i] = \mathsf{WriteReturn}_i(d, a)$ leads to $\mathsf{WriteReturn}_i(d, a)$.

Assumptions *L1* and *L2* force queues to be flushed; *L3* forces a ReadReturn response to processors when such a reply to an outstanding ReadRequest is enabled, and *L4* brings entries into the cache (via the In queue) on a miss. Similarly to *L3*, assumption *L5* forces a WriteReturn response to processors when such a reply to an outstanding WriteRequest is enabled.

Thus, assumption $L5$ is enough to ensure that every WriteRequest has a reply. Replies to ReadRequest's may depend on $L1$ and $L2$ to flush starred entries from the local queues, on $L3$ and again $L1$ to bring missing entries into the cache on a miss, and $L4$ to force a reply. Under these restrictions, the M_{cache} algorithm eventually responds to every Request unless the Request is overwritten.

Proposition 9 *In any execution α of M^h_{cache} satisfying liveness assumptions L1 through L5, $\alpha|P_i$ is either infinite or ends in a* Return *event.*

6.2 Infinite Simulations

Corollary 8 shows that finite executions of M_{cache} satisfy the constraints of Definition 2. If an item is allowed to stay on a queue forever, it is possible to construct an infinite execution of M_{cache} which does not satisfy Definition 2. (Consider a single write to an address a, followed by infinitely many reads that see the initial value of a.) The proof of Theorem 1 breaks down because the partial order constructed may not be a well-ordering. (That is, an event may have infinitely many predecessors. For the example above, the write is preceded by infinitely many reads.) Under liveness assumptions L1 and L2, which force items through the queues, the correctness condition applies to infinite executions of M_{cache}. (The other three liveness assumptions are only necessary to force replies to processors–if no replies occur, the sequence of events at that processor is either finite or an infinite sequence of overwritten requests.)

Proposition 10 *In any execution α of M^h_{cache} satisfying liveness assumptions L1 and L2, every* WriteReturn *event gets a sequence number. Furthermore, if infinitely many* ReadReturn *events acquire the same sequence number t, then t is the largest sequence number of all the* WriteReturn *events (of which there are only finitely many).*

Proposition 10 suffices to show that for executions of M_{cache}, the partial order constructed in the proof of Theorem 1 is a well-ordering. This observation in turn is the key to the following extension of Corollary 8 to the infinite case.

Corollary 11 *If α is an infinite execution of M_{cache} with behavior β, and α satisfies the liveness assumptions L1 and L2, then there exists a behavior β_s of M_{serial} such that $\beta_s|P_i = \beta|P_i$, $\forall i \in \{1, ..., n\}$.*

6.3 Global Time

As we observed earlier, processors with access to an accurate global clock could observe timing anomalies in interactions with M_{cache}. That is, an event occurring at time t_1 at processor P_i may be serialized after some event of processor P_j that occurred at time t_2, even though $t_1 < t_2$. However, assuming some bounds on the delay incurred by entries in the *In* and *Out* queues, it is possible to "blame" these timing anomalies on inaccuracies in the processors' observations of global time.

Theorem 2 *Let $\alpha = s_0\pi_1 s_1\pi_2 s_2...$ be an execution of M_{cache}, with $\beta = beh(\alpha)$, and let $t_1, t_2, ...$ be a non-decreasing sequence of real numbers, denoting the (global) times at which the corresponding events $\pi_1, \pi_2, ...$ occur. (If α is infinite, this sequence must be unbounded.) Suppose further that in α, no entry remains in any single queue for more than δ units of global clock time. Then there exists a corresponding behavior $\beta_s = \pi_{k_1}, \pi_{k_2}, ...$ of M_{serial} such that*

1. *for all processors q, $\beta|P_q = \beta_s|P_q$, and*
2. *$i < j$ implies that $t_{k_i} \leq t_{k_j} + 2\delta$.*

Theorem 2 may be restated as follows: assuming that the delay of each in-queue and of each out-queue is bounded by δ units of time then every execution of the M_{cache} system in which processors have perfectly synchronized local clocks corresponds (in the same sense as Definition 2) to an execution of an M_{serial} system in which their clocks may be 2δ apart.

Clearly, this generalizes to the case of imperfect local clocks in the M_{cache} system, which simulates an M_{serial} system in which clocks may diverge by up to an additional 2δ time units. Thus, there is a trade-off between queueing delays in the implementation of M_{cache} and the accuracy of local clocks.[§]

Proof: (Outline) Associate a new time, t'_i, with every event π_i in β, according to the following procedure:

[§]This discussion argues that synchronization of local clocks appears to degrade by 2δ time units in the M_{cache} system, as measured by the difference between two processor's local clocks at the same instant of (simulated) time. Other potentially important properties of local clocks, such as a bounds on their relative rates, may not be preserved.

- If π_i is a WriteRequest event, then $t'_i = t_i$.
- If π_i is a WriteReturn event, then t'_i is the time t_j of the corresponding MemoryWrite event π_j in α, if such an event exists. Otherwise, it is the maximum of t_0, $t_i + \delta$, and the time t_j of the last MemoryWrite event π_j in α, if such an event exists.
- If π_i is a ReadReturn_q event, then t'_i is the maximum of $t_i - \delta$ and t_j, where π_j is the MemoryWrite event corresponding to the last CacheUpdate_q event preceding π_i in α. (If no such MemoryWrite event exists, then $t'_i = t_i$.)
- If π_i is a ReadRequest_q event, then if π_i is followed in $\beta|P_q$ by a ReadReturn_q event π_j, t'_i is the minimum of t_i and t'_j, and $t'_i = t_i$ otherwise.

We make two claims, the proofs of which are a technical exercise we omit. (The first is straightforward–the second is a technical extension of the proof of Theorem 1.)

Claim 12 $t_i - \delta \leq t'_i \leq t_i + \delta$

Claim 13 *The natural ordering of the π_i induced by the t'_i's is a behavior β_s satisfying the first property of the theorem.*

Effectively, writes are moved up to δ time units forward in time, and reads are moved at most δ time units backward. ∎

6.4 A Non-triviality Result

Under reasonable liveness assumptions, we have argued that M_{cache} replies to every request that is not overwritten by another request. Furthermore, we argued that under these same assumptions, all the behaviors of M_{cache} satisfy Definition 2. However, both properties are trivially true of M_{serial}, as well. Is it possible to argue formally that M_{cache} is an improvement over M_{serial}? We have some partial results.

First, it is possible to define a simple mapping from the states of M_{serial} to those of M_{cache} and show that every behavior of M_{serial} is also a behavior of M_{cache}. (This is the "possibilities mapping" technique of [LT87, Lyn88], closely related to bisimulation [Mil80] or refinement mappings [AL88].) So, in some sense, M_{cache} is no worse than M_{serial}, in that it permits as many histories at the interface to the processors. It is also easy to construct examples that demonstrate that M_{cache} permits strictly more histories than does M_{serial}. However, we know of no characterization of the behaviors of M_{cache} that is much simpler than the description of M_{cache} itself. Furthermore, it is possible to generate sequences satisfying Definition 2 that are not behaviors of M_{cache}, e.g., assuming d_{init}, d_1 and d_2 are distinct:

1	$\vdash \mathsf{Write}(d_1, 1) \dashv$	$\vdash \mathsf{Read}(d_1, 1) \dashv$		
2			$\vdash \mathsf{Write}(d_2, 1) \dashv$	$\vdash \mathsf{Read}(d_1, 1) \dashv$

A corresponding serial behavior that satisfies definition 2 is:

1		$\vdash \mathsf{Write}(d_1, 1) \dashv$	$\vdash \mathsf{Read}(d_1, 1) \dashv$	
2	$\vdash \mathsf{Write}(d_2, 1) \dashv$			$\vdash \mathsf{Read}(d_1, 1) \dashv$

So we do not have a "completeness" result, in the sense of capturing all the behaviors permitted by Definition 2.

It is easy to define an automaton satisfying such a completeness result: simply record the entire history in the automaton state, and enable an action exactly if it preserves the constraint of Definition 2. However, it is hard to see how such an automaton could be built, i.e., implemented practically, without restricting its behavior. Our intention in designing M_{cache} was to describe an algorithm whose implementation would be straightforward. To reason precisely about such implementations would require formalization of particular technologies, an important problem beyond the scope of this paper.

7 Discussion

We hope this work will have an impact beyond the particular contributions of the formalization of sequential consistency and the lazy cache algorithm, in fostering a fuller exploration of both shared-memory consistency conditions and of algorithms supporting them. No particular consistency condition is "correct", beyond the scope of a particular application. (For example, the timing anomalies discussed in Section 6 have serious implications for real-time programming on sequentially consistent systems.) Similarly, no particular algorithm may be optimal for all applications and hardware technologies. But within any particular context, it is worthwhile to formalize the constraints imposed on the shared memory and search for the weakest correctness condition consistent with them. This in turn admits the widest possible range of implementations.

The drive to construct efficient large-scale multiprocessors has led architects to implement consistency conditions weaker than sequential consistency. (E.g., [GW88, LS88, HA90].) Precise formulation of such weaker correctness conditions will be important in writing and validating software for such systems. Indeed, the success of such efforts may depend upon finding natural characterizations of these conditions that are more succinct than a straightforward description of the hardware.

Section 6.3 discussed an interesting trade-off between allowable queueing delays in the cache algorithm and perceived accuracy of local clocks. Other design trade-offs may also be waiting for discovery.

The correctness condition we introduce is appropriate for systems in which processors interact *only* through the shared memory. If processors have an alternative means of communication, they may detect discrepancies during execution if they communicate at times when their local caches represent different images of the shared memory. Following [Bro90], the lazy cache algorithm can be extended to deal with such systems by requiring the communicating processors to bring their caches up to date before participating in such communication. For example, this action could be taken as part of interrupt handling.

Interaction with users or other agents external to the system represent another implicit communication channel between processors. Correct system behavior at this interface is an additional goal of the system specification. Users of a multiprocessor system running the lazy cache algorithm and that communicate through other channels than the cache may observe anomalous behavior. As above, this may be avoided by requiring each processor to have an up-to-date cache before interacting with the external world. The effects of communication (synchronization) via a global clock were considered in Section 6.3.

The system model may be extended to accommodate these changes by making external actions an explicit part of each processor's interface, and requiring the processor to perform a handshake with the memory system prior to taking such external action. The memory system would flush the local cache queues before responding to the handshake. Using a natural notion of component composition due to Lynch and Tuttle [LT87], and extending the proof of Theorem 1, it is possible to argue that each sequence of external signals produced by a system running this modified lazy cache algorithm is a sequence that could be produced in a system running a conventional shared memory. (Similar modifications can be made to extend the interface of the memory itself to include more powerful operations such as test-and-set.)

Finally, this research raises many interesting implementation and performance issues. If queues are to be implemented in hardware, they must be bounded. How can they be managed so as to avoid overflow? For particular performance parameters, how large should the hardware queues be? Can the greater concurrency of the lazy cache algorithm translate into real performance gains? The $\mathsf{MemoryWrite}$ action, which atomically updates n queues, is a point of global synchronization that we envision being implemented via a bus. Is there a more efficient way to utilize a bus? Could multi-bus architectures exploit a more concurrent algorithm, while supporting the same consistency condition? Can shared-memory be implemented efficiently on hardware supporting other kinds of communication, such as point-to-point connections? We believe the methodology and perspective of this paper will contribute to answering some of these questions.

8 Acknowledgement

The authors thank the anonymous referees, whose careful reading and detailed criticisms were of great value.

References

[ABHN91] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal memory. Technical Report GIT-CC-91/42, Georgia Institute of Technology, September 1991.

[AH90a] S.V. Adve and M.D. Hill. Implementing sequential consistency in cache-based systems. In *1990 International Conference on Parallel Processing*, pages I–47–I–50, August 1990.

[AH90b] S.V. Adve and M.D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 165–175, July 1988. Edinburgh, Scotland. Also available as a Digital Systems Research Center technical report, 130 Lytton Avenue, Palo Alto, CA 94301.

[Bro90] Geoffrey Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–36, 1990.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27:1112–1118, 1978.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *13th International Symposium on Computer Architecture*, pages 434–442, 1986.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GMG91] P.B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Symposium on Parallel Algorithms and Architectures*, July 1991.

[Goo87] J. Goodman. Coherency for multiprocessor virtual address caches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 72–81, October 1987.

[GW88] J. Goodman and P. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *15th International Symposium on Computer Architecture*, pages 422–431, June 1988.

[HA90] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–309. IEEE Computer Society Press, 1990.

[KEW+85] R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

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

[LS88] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

[Lyn88] N. Lynch. I/O automata: A model for discrete event systems. In *22nd Annual Conference on Information Science and Systems*. Princeton University, March 1988. Also MIT technical report number MIT/LCS/TM-351.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980. Lecture Notes in Computer Science 92.

[Mis86] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[SD87] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *14th International Symposium on Computer Architecture*, pages 234–243, 1987.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14:473–540, 1982.

[SS86] P. Sweazey and A. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *13th International Symposium on Computer Architecture*, pages 414–423, june 1986.

[SS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *Transaction on Programming Languages and Systems*, 10(2):282–312, 1988.

[YYF85] W. Yen, D. Yen, and K. Fu. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, c-34(1):56–65, January 1985.

A Proof of Lemma 7

Let $<_{\alpha P}$ be a relation satisfying condition (1) of $<_\alpha$ and $<_{\alpha WR}$ be a relation satisfying conditions (2) and (3) of $<_\alpha$.

Proposition 14 *Both $<_{\alpha P}$ and $<_{\alpha WR}$ are transitive and irreflexive.*

Lemma 15 *For all events π and ϕ in α, $(\pi <_{\alpha P} \phi) \Rightarrow \neg(\phi <_{\alpha WR} \pi)$.*

Proof: The proof is by induction; it suffices to assume that $\alpha = \alpha'\eta s$, for action η and state s, and that for all events π and ϕ in α', $(\pi <_{\alpha' P} \phi) \Rightarrow \neg(\phi <_{\alpha' WR} \pi)$. The proof is by case analysis on action η. The only cases that are not immediate are the following:

- Case: $\eta = \mathsf{MemoryWrite}_i(d, a)$
  By Proposition 3, there is a $\mathsf{WriteReturn}_i(d, a)$ action ϕ in α' such that $WC_{\alpha'}(\phi) = \infty$. Further, $WC_{\alpha'}(\psi) < \infty$ on all $\mathsf{WriteReturn}_i$ and $\mathsf{ReadReturn}_i$ actions ψ preceding ϕ in α', and $WC_{\alpha'}(\psi) = \infty$ on any $\mathsf{WriteReturn}_i$ actions ψ that follow ϕ (no $\mathsf{ReadReturn}_i$ actions follow ϕ.) Also, by Lemma 2, $WC_\alpha(\phi)$ is strictly greater than any integer value (i.e. $\neq \infty$) of $WC_{\alpha'}$ on any argument. Then $<_{\alpha WR}$ differs from $<_{\alpha' WR}$ in that it orders ϕ before the other $\mathsf{WriteReturn}$ events ψ in α' such that $WC_{\alpha'}(\psi) = \infty$. For each of these actions, $\neg(\psi <_{\alpha' P} \phi)$ holds. Since $<_{\alpha' P} = <_{\alpha P}$, the lemma follows.

- Case: $\eta = \mathsf{WriteReturn}_i(d, a)$
  Then $<_{\alpha WR}$ differs from $<_{\alpha' WR}$ by the addition of a new maximal element, since $WC_\alpha(\eta) = \infty$. Similarly, $<_{\alpha P}$ differs from $<_{\alpha' P}$ by the addition of a new maximal element since η is ordered after all other operations of P_i in α. Because $\eta <_{\alpha P} \phi$ and $\eta <_{\alpha WR} \phi$ are false for all actions ϕ, the lemma follows.

- Case: $\eta = \mathsf{ReadReturn}_i(d, a)$
  The relation $<_{\alpha WR}$ differs from $<_{\alpha' WR}$ in that it orders η with respect to the actions in α'. From Lemma 2, $WC_\alpha(\eta)$ is no less than the value of WC_α on all other events of P_i (because $lastin_i \geq lastout_i$ and both are strictly increasing), thus the lemma follows.

∎

In the following discussion we use the notation $\pi <_\alpha^n \phi$ to denote

$(\exists a_1 ... a_{n-1} : \pi <_\alpha a_1 \wedge a_1 <_\alpha a_2 \wedge ... \wedge a_{n-1} <_\alpha \phi)$

Lemma 16 *$<_\alpha \cup <_\alpha^2 \cup <_\alpha^3$ is the transitive closure of $<_\alpha$.*

Proof: It suffices to show that

$(\forall n : n > 3 : \pi <_\alpha^n \phi \Rightarrow (\exists m : m < n : \pi <_\alpha^m \phi))$

¿From Proposition 14 we conclude

$a_0 <_{\alpha P} a_1 \;\wedge\; a_1 <_{\alpha P} a_2 \;\;\Rightarrow\;\; a_0 <_{\alpha P} a_2$, and

$a_0 <_{\alpha WR} a_1 \;\wedge\; a_1 <_{\alpha WR} a_2 \;\;\Rightarrow\;\; a_0 <_{\alpha WR} a_2$

We now show

$a_0 <_{\alpha WR} a_1 \;\wedge\; a_1 <_{\alpha P} a_2 \;\wedge\; a_2 <_{\alpha WR} a_3 \;\;\Rightarrow\;\; a_0 <_{\alpha WR} a_3$

¿From Lemma 15, it is not the case that $a_2 <_{\alpha WR} a_1$. If $<_{\alpha WR}$ does not order a_1 and a_2, then by the definition of $<_{\alpha WR}$, $WC_\alpha(a_1) = WC_\alpha(a_2)$ and both are $\mathsf{WriteReturn}$ actions. Proposition 3 implies that $WC_\alpha(a_2) = \infty$, contradicting $a_2 <_{\alpha WR} a_3$. It follows that $a_1 <_{\alpha WR} a_2$, and an appeal to Proposition 16 proves the lemma.

Since all "chains" of length greater than 3 must contain a "subchain" satisfying one of the three proceeding properties, the lemma follows.

∎

Proof (Lemma 7): We now show that the transitive closure of $<_\alpha$ is irreflexive, i.e., $<_\alpha$, $<_\alpha^2$, and $<_\alpha^3$ are irreflexive.

¿From Proposition 14 and Lemma 15 we conclude that $<_\alpha$ and $<_\alpha^2$ are irreflexive. We prove that $<_\alpha^3$ is irreflexive by contradiction. ¿From our proof of Lemma 16, we conclude that it is sufficient to consider reflexive "chains" of the form

$a_0 <_{\alpha P} a_1 \;\wedge\; a_1 <_{\alpha WR} a_2 \;\wedge\; a_2 <_{\alpha P} a_0$

If this chain is in $<^3_\alpha$, then from the transitivity of $<_{\alpha P}$,

$a_2 <_{\alpha P} a_1 \ \wedge \ a_1 <_{\alpha WR} a_2$

is in $<^2_\alpha$. Since, $<^2_\alpha$ is irreflexive, $<^3_\alpha$ is irreflexive. ∎