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 Descriptors: B.3 2 [**Memory Structures**]. Design Styles—*cache memories*; *shared memory*; C.1.2. [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors (MIMD)*; *parallel processors*; D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*

General Term: Design

Additional Key Words and Phrases: Cache coherence, sequential consistency, shared memory

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 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

A preliminary version of this paper appeared in the *Proceedings of the 1988 ACM Symposium on Parallel Architectures and Algorithms* (Sante Fe, N.M., June). ACM, New York, 1989, pp. 209–222. 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, Cornell University, Ithaca, NY 14853-5401; M. Merritt, 600 Mountain Avenue, Murray Hill, NJ 07974.

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 [14]. Typically, cache consistency algorithms implement much stronger conditions. For example, Lamport himself considers only algorithms satisfying stronger conditions than sequential consistency [14]. 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 that have accurate measures of the passage of time, and we discuss a trade-off between the observed accuracies of local clocks and queuing 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 that, 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 [22, 23]. Although 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 multiprocess 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 nonconventional 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 "complete" 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 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.

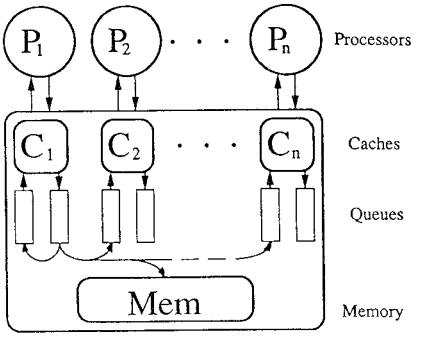


Fig. 1. The architecture of the lazy cache algorithm.

1.1 Previous Work

As noted above, the consistency definition explored in this paper is a formalization of *sequential consistency* originally discussed by Lamport [14]: "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" [6]. 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 [20]. For systems in which this definition has a natural interpretation, coherence is usually the correctness condition cited. (See, e.g., [10], [13], [22]–[24]). 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.)

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 [7]. 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. Although applicable to a broader class of architectures, like cache coherence the definition of strong ordering is somewhat informal and makes implicit

architectural assumptions.[1] 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 [20] 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*." [20]. 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 [5]. 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 on 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 [21] explores a similar weak consistency condition in the off-line context of compiling for shared-memory multiprocessors. Conflict relations are extracted from the complete program text, and minimal synchronization conditions are found. This work complements the on-line approach taken in this 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 proved correct. Liveness and timing properties are addressed in Section 6. The paper concludes with a discussion and additional problems.

2. MEMORY SYSTEMS

Following [16] and [17], a system is defined as a set of states and a set of actions that are multivalued functions from states to states. A system gives

[1]Recently, Adve and Hill have shown that strongly ordered memories are not, in general, sequentially consistent [3].

rise to a set of *executions*, which are finite or infinite sequences $s_0, \pi_1, s_1, \ldots$ of alternating states (the s_is) and actions (the π_is), 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 a 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:

(1) **ReadRequest**$_i(a)$ (processor i requests the data value in address a);
(2) **ReadReturn**$_i(d, a)$ (data d from address a returned to processor i);
(3) **WriteRequest**$_i(d, a)$ (processor i requests that data d be written to address a); and
(4) **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, **Read**(d, a) and **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:

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

The array *Handshake* stores outstanding memory operations: *Handshake*$[i]$ = **WriteRequest**(d, a) indicates an outstanding write request by processor i of value d to memory location a, and *Handshake*$[i]$ = **WriteReturn**(d, a) indicates that the write operation has occurred and enables the response to processor i. Similarly, *Handshake*$[i]$ = **ReadRequest**(a) indicates an outstanding read request by processor i of memory location a, and *Handshake*$[i]$ = **ReadReturn**(d, a) indicates that 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:

ReadRequest$_i(a)$:: *Handshake*$[i]$:= **ReadRequest**$_i(a)$

Read$_i(d, a)$::
Mem$[a]$ = d → *Handshake*$[i]$:= **ReadReturn**$_i(d, a)$
∧*Handshake*$[i]$ = **ReadRequest**$_i(a)$

ReadReturn$_i(d, a)$::
Handshake$[i]$ = **ReadReturn**$_i(d, a)$ → *Handshake*$[i]$:= *null*

WriteRequest$_i(d, a)$ *Handshake*$[i]$:= **WriteRequest**$_i(d, a)$

Write$_i(d, a)$::
Handshake$[i]$ = **WriteRequest**$_i(d, a)$ → *Mem*$[a]$:= d
Handshake$[i]$:= **WriteReturn**$_i(d, a)$

WriteReturn$_i(d, a)$::
Handshake$[i]$ = **WriteReturn**$_i(d,a)$ → *Handshake*$[i]$:= *null*

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

Thus, **ReadRequest**$_i(a)$ can be executed from any state and has the effect of recording a request to read from address a; **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 **ReadReturn**$_i(d, a)$, can only be executed in a state where *Handshake*$[i]$ = **ReadReturn**$_i(d, a)$, indicating that a read operation for i has occurred. Similarly, **WriteRequest**$_i(d, a)$ can be executed from any state and has the effect of recording a write request; **WriteReturn**$_i(d, a)$ can only be executed from a state in which *Handshake*$[i]$ = **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 **WriteReturn**$_i(d, a)$; and **WriteReturn**$_i(d, a)$ can only be executed in a state where *Handshake*$[i]$ = **WriteReturn**$_i(d, a)$, indicating that a write of d to a for i has occurred.[2]

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 {**ReadRequest**$_i(a)$, **ReadReturn**$_i(d, a)$,

[2]The external actions of M_{shared} model a handshake between each processor and the shared memory; the **ReadRequest**$_i$ and **WriteRequest**$_i$ actions model inputs from processor i to the memory, and the **ReadReturn**$_i$ and **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.

$\mathbf{WriteRequest}_i(d, a), \mathbf{WriteReturn}_i(d, a) : d \in data, a \in address\}$. 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, $\beta| \cup_i P_i$ 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* of 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* [12]. (See, also, [19] for an 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 exists 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 from 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.[3]

In particular, Definition 2 requires that there exists 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 that $d \neq d_{init}$ where d_{init} is the initial value of $Mem[1]$:

$$\begin{array}{lll} \mathbf{WriteRequest}_1(d,1), & \mathbf{Write}_1(d,1), & \mathbf{WriteReturn}_1(d,1), \\ \mathbf{ReadRequest}_2(1), & \mathbf{Read}_2(d_{init},1), & \mathbf{ReadReturn}_2(d_{init},1). \end{array}$$

Such examples are probably more easily understood when described graphically, as below:

$$\begin{array}{ll} 1 & \vdash \mathbf{W}(d,1) \dashv \\ 2 & \qquad\qquad \vdash \mathbf{R}(d_{init},1) \dashv \end{array}$$

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

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

$$\begin{array}{lll} \mathbf{ReadRequest}_2(1), & \mathbf{Read}_2(d_{init},1), & \mathbf{ReadReturn}_2(d_{init},1), \\ \mathbf{WriteRequest}_1(d,1), & \mathbf{Write}_1(d,1), & \mathbf{WriteReturn}_1(d,1), \end{array}$$

and, graphically,

$$\begin{array}{ll} 1 & \qquad\qquad \vdash \mathbf{W}(d,1) \dashv \\ 2 & \vdash \mathbf{R}(d_{init},1) \dashv \end{array}$$

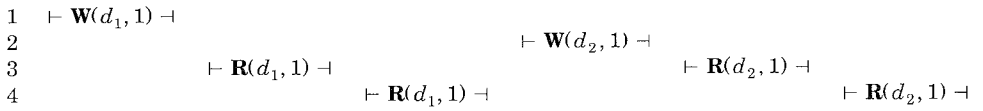
Thus, the read and write operations of processors 1 and 2 occur in different orders in the simulated behavior than in the actual behavior.

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

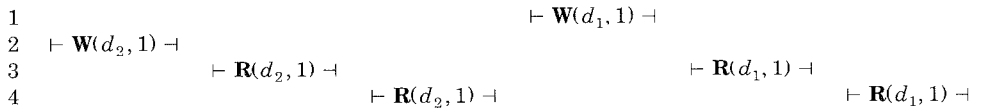
$$\begin{array}{ll} 1 & \vdash \mathbf{W}(d_1,1) \dashv \\ 2 & \qquad\qquad \vdash \mathbf{W}(d_2,1) \dashv \\ 3 & \qquad\qquad\qquad\qquad \vdash \mathbf{R}(d_1,1) \dashv \quad \vdash \mathbf{R}(d_2,1) \dashv \\ 4 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdash \mathbf{R}(d_2,1) \dashv \quad \vdash \mathbf{R}(d_1,1) \dashv \end{array}$$

[3] 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, [2], [4], [8], [9], [12], and [15]. These approaches all involve some complication of the programming model, while sequential consistency allows the programmer to assume that the memory is serial.

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



Meanwhile, the following behavior of M_{serial} preserves the order of actions of processor 4:



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 (Definition 2) 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 Section 4 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 $\mathbf{WriteReturn}_i(d, a)$ by processor i appends the pair (d, a) to Out_i. As a separate atomic $\mathbf{MemoryWrite}_i$ 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 $\mathbf{CacheUpdate}_i$ action use the entries 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 the In_i queue into C_i.

As Theorem 8 below demonstrates, this restriction is sufficient to guarantee that the finite behaviors of M_{cache} satisfy Definition 2. Corollary 12 argues that, under liveness assumptions prohibiting infinite queuing 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 read ($\mathbf{ReadReturn}_i's$) 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: $\mathbf{MemoryRead}_i$ actions add any $(data, address)$ pair from *Mem* to In_i, and $\mathbf{CacheInvalidate}_i$ 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 and 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 of 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* × *Data*. These functions may be updated via two operations:
  —*update*(C_i, d, a) returns the partial function that maps address a to value d, but is otherwise identical to C_i.
  —*restrict*(C_i) 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 $C_i \subseteq Mem, \forall i \in \{1, \ldots, n\}$.

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

ReadRequest$_i(a)$::
  Handshake[i] = **ReadRequest**$_i(a)$

ReadReturn$_i(d, a)$::
  Handshake[i] = **ReadRequest**$_i(a)$
  $\wedge C_i[a] = d$
  $\wedge Out_i = \{\ \}$
  $\wedge$ (There are no starred entries in In_i.)
  → *Handshake*[i] = *null*

WriteRequest$_i(d, a)$::
  Handshake[i] := **WriteRequest**$_i(d,a)$

WriteReturn$_i(d, a)$::
  Handshake[i] = **WriteRequest**$_i(d, a)$
  → *Handshake*[i] = *null*;
  Out_i := *append*(Out_i, (d, a))

MemoryWrite$_i(d, a)$::
  head(Out_i) = (d, a)
  → *Mem*[a] := d;
  Out_i = *tail*(Out_i);
  $(\forall k \neq i \| :: In_k$:= *append*(In_k, (d, a)))
  (In_i = *append*(In_i, $(d, a, *)$))

MemoryRead$_i(d, a)$::
  Mem[a] = d
  → In_i := *append*(In_i, (d, a))

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

CacheInvalidate$_i$:: $C_i := restrict(C_i)$

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

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 $\beta|P_i = \beta_s|P_i, \forall i \in \{1, \ldots, n\}$.

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 $Lastin_i, \forall i \in \{1, \ldots, n\}$, of type integer, which record the sequence number of the last update applied to C_i;

—n variables $Lastout_i, \forall i \in \{1, \ldots, n\}$, of type integer, which record the sequence number of the last write by P_i applied to *Mem*; and

—the write entries of the queues $In_1, \ldots, In_n$, 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$_i(d, a)$::
$head(Out_\iota) = (d, a)$ → $Mem[a] := d$;
$WriteCounter := WriteCounter + 1$;
$LastOut_\iota := WriteCounter$;
$Out_\iota := tail(Out_\iota)$;
$(\forall k \neq \iota \|::$
$In_k := append(In_k, d, a, WriteCounter)))$;
$In_\iota := append(In_\iota, (d, a, *, WriteCounter))$;
$Hist_M := append(Hist_M, Mem)$

MemoryRead$_i(d, a)$::
$Mem[a] = d$ → $In_\iota := append(In_\iota, (d, a, WriteCounter))$

CacheUpdate$_i(d, a)$::
$head(In_\iota)$ is either (d, a, wc) or $(d, a, *, wc)$, for some wc. → $In_\iota := tail(In_\iota)$;
$C_\iota := update(C_\iota, d, a)$;
$Lastin_\iota := wc$

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

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}, $(\forall i \in \{1, \ldots, n\})$:*

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

PROOF. Proving the invariants requires checking that they are maintained by the nine actions of M^h_{cache}. For example, consider the action **MemoryWrite**$_i$. This action appends the modified *Mem* to $Hist_m$ and increments *WriteCounter* preserving (1); assigns *WriteCounter* to $LastOut_\iota$ 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 $\mathbf{MemoryWrite}_i$ 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 $\beta_s|P_i = \beta|P_i, \forall i \in \{1, \ldots, n\}$. We do so by looking at an execution α of M^h_{cache} with behavior β and by extracting an important ordering relationship between events in β. The sequences that are candidates for the simulated serial behavior β_s are shown to be those that extend this ordering relationship.

Given any state of an execution of α of M^h_{cache}, each entry (d, a) in queue Out_i was originally inserted as the consequence of a particular $\mathbf{WriteReturn}_i(d, a)$ event. When an entry (d, a) in Out_i is dequeued (by a $\mathbf{MemoryWrite}_i(d, a)$ 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 have their 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 $\{1, 2, \ldots, WriteCounter\} \cup \{q | q$ 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 $\mathbf{WriteReturn}_i$ 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:

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 $\{1, 2, \ldots, WriteCounter\} \cup \{q | q$ is an entry in Out_i for some $i\}$.*

In addition, a second function WC^R_α maps the **ReadReturn** events to the integers $\{0, 1, \ldots, WriteCounter\}$, where $WC^R_\alpha(\mathbf{ReadReturn}_i)$ is the value of $Lastin_i$ at the time that $\mathbf{ReadReturn}_i$ 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 $\{0, 1, \ldots\} \cup \{\infty\}$, 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 **WriteReturn**s in the simulated serial behavior and corresponds to the order of 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 that have already returned to

the calling processors, 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 if π precedes ϕ in α, then $\pi <_\alpha \phi$.
(2) If π and ϕ are **WriteReturn** or **ReadReturn** events and if $WC_\alpha(\pi) < WC_\alpha(\phi)$, 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 8. *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 $\beta_s|P_i = \beta|P_i, \forall i \in \{1, \ldots, n\}$.*

PROOF. By Lemma 7, a reordering β_s of the events in β that is consistent with $<_\alpha$ exists. The equality of $\beta_s|P_i = \beta|P_i$ 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 of $Hist_M[0]$). 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_v^x to denote the state obtained from state r by assigning to state variable x the value v):

—$\pi = \mathbf{ReadRequest}_i(a)$: $\gamma = \mathbf{ReadRequest}_i(a) r^{Handshake[i]}_{\mathbf{ReadRequest}_i(a)}$.

—$\pi = \mathbf{ReadReturn}_i(d, a)$: $\gamma = \mathbf{Read}_i(d, a) r^{Handshake[i]}_{\mathbf{ReadReturn}_i(d,a)}$ $\mathbf{ReadReturn}_i(d, a) r^{Handshake[i]}_{null}$. It remains to show that action $\mathbf{ReadReturn}_i(d, a)$ is enabled in state r, that is $Mem[a] = d$ and $Handshake[i] = \mathbf{ReadRequest}_i(a)$, and that $Mem = Hist_M[WC_\alpha(\pi)]$ in state r. From Proposition 5 and the definition of $<_\alpha$, $\mathbf{ReadRequest}_i(a)$ 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 the invariant preserved by the construction of α_s.

—$\pi = \mathbf{WriteRequest}_i(d, a)$: $\gamma = \mathbf{WriteRequest}_i(d, a) r^{Handshake[i]}_{\mathbf{WriteRequest}_i(d,a)}$.

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

From Proposition 4, $Handshake[i] = \mathbf{WriteRequest}_i(d, a)$ 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 9. *If α is a finite execution of M_{cache} with behavior β, then there exists a behavior β_s of M_{serial} such that $\beta_s|P_i = \beta|Pi, \forall i \in \{1, \ldots, n\}$.*

PROOF. Immediate from Proposition 1 and Theorem 8. □

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 states that there are behaviors satisfying Definition 2 that cannot be produced by the lazy cache algorithm.

6.1 Liveness

We define the following *liveness constraints* on executions 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 $\mathbf{CacheUpdate}_i$.

(L2) $Out_i \neq \{\ \}$ leads to $\mathbf{MemoryWrite}_i$.

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

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

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

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 on (L4) to force a reply. Under these restrictions, the M_{cache} algorithm eventually responds to every **Request** unless the **Request** is overwritten.

PROPOSITION 10. *In any execution α of M^h_{cache} satisfying liveness assumptions* (L1)–(L5), *$\alpha|P_i$ is either infinite or ends in a* **Return** *event.*

6.2 Infinite Simulations

Corollary 9 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} that 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 8 breaks down because the partial order constructed may not be a well-ordering. (I.e., 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 11. *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 of the* **WriteReturn** *events* (*of which there are only finitely many*).

Proposition 11 suffices to show that for executions of M_{cache} the partial order constructed in the proof of Theorem 8 is a well-ordering. This observation in turn is the key to the following extension of Corollary 9 to the infinite

case:

COROLLARY 12. *If α is an infinite execution of M_{cache} with behavior β and if α satisfies the liveness assumptions (L1) and (L2), then there exists a behavior β_s of M_{serial} such that $\beta_s|P_i = \beta|Pi, \forall i \in \{1, \ldots, n\}$.*

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 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 13. *Let $\alpha = s_0\pi_1 s_1 \pi_2 s_2 \cdots$ be an execution of M_{cache}, with $\beta = beh(\alpha)$, and let $t_1 t_2, \ldots$ be a nondecreasing sequence of real numbers, denoting the (global) times at which the corresponding events $\pi_1, \pi_2, \ldots$ 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 $\beta_s = \pi_{k_1}, \pi_{k_2}, \ldots$ 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 13 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 up to an additional 2δ time units. Thus, there is a trade-off between queuing delays in the implementation of M_{cache} and the accuracy of local clocks.[4]

PROOF. (Outline) Associate a new time, t_i', with every event π_i in β, according to the following procedure:

—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.

[4] This discussion argues that synchronizaton 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 **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** events exists, then $t_d' = t_i$.)

—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 8.)

Claim 14. $t_i - \delta \leq t_i' \leq t_i + \delta$.

Claim 15. 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 Nontriviality Result

Under reasonable liveness assumptions, we have argued that M_{cache} replies to every request that is not overwritten by another request. Furthermore, we have argued that, under these same assumptions, all of the behaviors of M_{cache} satisfy Definition 2. However, both properties are trivially true of M_{serial} as well. It is 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 to show that every behavior of M_{serial} is also a behavior of M_{cache}. (This is the "possibilities mapping" technique of [16] and [17], closely related to bisimulation [18] or refinement mappings [1]). 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}, for example, by assuming that d_{init}, d_1, and d_2 are distinct:

1	$\vdash \mathbf{W}(d_1, 1) \dashv$	$\vdash \mathbf{R}(d_1, 1) \dashv$		
2			$\vdash \mathbf{W}(d_2, 1) \dashv$	$\vdash \mathbf{R}(d_1, 1) \dashv$

A corresponding serial behavior that satisfies Definition 2 is the following:

1		$\vdash \mathbf{W}(d_1, 1) \dashv$	$\vdash \mathbf{R}(d_1, 1) \dashv$	
2	$\vdash \mathbf{W}(d_2, 1) \dashv$			$\vdash \mathbf{R}(d_1, 1)$

So we do not have a "completeness" result, in the sense of capturing all of 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; that is, 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 the algorithms supporting them. No particular consistency condition is "correct," beyond the scope of a particular application. (E.g., 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 to 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., [11], [12], and [15]). 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 on finding natural characterizations of these conditions that are more succinct than a straight forward description of the hardware.

Section 6.3 has discussed an interesting trade-off between allowable queuing 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 [5], 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 have been 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 by 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 [17] and extending the proof of Theorem 8, 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 **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 multibus 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.

APPENDIX. Proof of Lemma 7

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

PROPOSITION 16. Both $<_{\alpha P}$ and $<_{\alpha WR}$ are transitive and irreflexive.

LEMMA 17. *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. $\eta =$ **MemoryWrite**$_i(d, a)$. By Proposition 3, there is a **WriteReturn**$_i(d, a)$ action ϕ in α' such that $WC_{\alpha'}(\phi) = \infty$. Furthermore, $WC_{\alpha'}(\psi) < \infty$ on all **WriteReturn**$_i$ and **ReadReturn**$_i$ actions ψ preceding ϕ in α', and $WC_{\alpha'}(\psi) = \infty$ on any **WriteReturn**$_i$ actions ψ that follow ϕ (no **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 **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. $\eta = \mathbf{WriteReturn}_i(d, a)$. 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. $\eta = \mathbf{ReadReturn}_i(d, a)$. 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

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

LEMMA 18. *$<_\alpha \cup <_\alpha^2 \cup <_\alpha^3$ 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 16 we conclude that

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

and

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

We now show that

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

From Lemma 17, 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 **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 18 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 know show that the transitive closure of $<_\alpha$ is irreflexive, that is, $<_\alpha$, $<_\alpha^2$, and $<_\alpha^3$ are irreflexive.

From Proposition 16 and Lemma 17, we conclude that $<_\alpha$ and $<_\alpha^2$ are irreflexive. We prove that $<_\alpha^3$ is irreflexive by contradiction. From our proof of Lemma 18, 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 $<_\alpha^3$, then from the transitivity of $<_{\alpha P}$

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

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

ACKNOWLEDGMENT

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

REFERENCES

1. ABADI, M., AND LAMPORT, L. The existence of refinement mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, (Edinburgh, Scotland, July). ACM, New York, 1988, pp. 165–175 (Also available as Tech. Rep., Digital Systems Research Center, Palo Alto, Calif.).
2. ADVE, S. V., AND HILL, M. D. Weak ordering—A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Wash., May 28–31). IEEE Computer Society, Los Angeles, 1990, pp. 2–14.
3. ADVE, S. V., AND HILL, M. D. Implementing sequential consistency in cache-based systems. In *1990 International Conference on Parallel Processing* (University Park, Penn., Aug 13–17). Pennsylvania State Univ., University Park, Penn., 1990, pp. I-47–I-50.
4. AHAMAD, M., BURNS, J. E., HUTTO, P. W., AND NEIGER, G. Causal memory. Tech. Rep. GIT-CC-91/42, Georgia Institute of Technology, College of Computing, Atlanta, Sept., 1991.
5. BROWN, G. Asynchronous multicaches. *Distrib. Comput. 4*, 1 (Mar. 1990), 31–36.
6. CENSIER, L. M., AND FEAUTRIER, P. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput. C-27* 12 (Dec. 1978), 1112–1118.
7. DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. Memory access buffering in multiprocessors. In *13th International Symposium on Computer Architecture* (Tokyo, June 2–5). IEEE Computer Society, 1986, pp. 434–442.
8. GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Wash , May 28–31). IEEE Computer Society, Los Angeles, 1990, pp. 15–26.
9. GIBBONS, P. B., MERRITT, M., AND GHARACHORLOO, K. Proving sequential consistency of high-performance shared memories. In *Symposium on Parallel Algorithms and Architectures* (Hilton Head, S. Carol., July 21–24). ACM, New York, 1991.
10. GOODMAN, J. Coherency for multiprocessor virtual address caches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)* (Palo Alto, Calif., Oct. 5–8) IEEE Computer Society, Los Angeles, 1987, pp. 72–81.
11. GOODMAN, J., AND WOEST, P. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *15th International Symposium on Computer Architecture* (Honolulu, Hawaii, May 30–June 2). IEEE Computer Society, Los Angeles, 1988, pp. 422–431
12. HUTTO, P., AND AHAMAD, M. Slow memory: Weakening consistency to enhance concurrency in distributed memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems* (Paris, France, May 28–June 1). IEEE, New York, 1990, pp. 302–309.
13. KATZ, R., EGGERS, S., WOOD, D., PERKINS, C., AND SHELDON, R. Implementing a cache consistency protocol. In *12th International Symposium on Computer Architecture* (Boston, June 17–19). 1985, pp. 276–283.
14. LAMPORT, L. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28*, 9 (Sept. 1979), 690–691.
15. LIPTON, R., AND SANDBERG, J. Pram: A scalable shared memory. Tech. Rep. CS-TR-180-88, Computer Science Dept., Princeton Univ., Princeton, N.J., Sept. 1988.

16. LYNCH, N. I/O automata: A model for discrete event systems. In *22nd Annual Conference on Information Science and Systems.* Princeton Univ., Princeton, N.J., Mar. 1988. (Also available as Tech. Rep. MIT/LCS/TM-351, Lab. for Computer Science, MIT, Cambridge, Mass.)
17. LYNCH, N., AND TUTTLE, M. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation* (Vancouver, Canada, Aug. 10–12). ACM, New York, 1987, pp. 137–151. (Expanded version available as Tech. Rep. MIT/LCS/TR-387, Laboratory for Computer Science, MIT, Cambridge, Mass., April 1987.)
18. MILNER, R. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York, 1980.
19. MISRA, J. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst. 8* 1 (Jan. 1986), 142–153.
20. SCHEURICH, C., AND DUBOIS, M. Correct memory operation of cache-based multiprocessors. In *14th International Symposium on Computer Architecture* (Pittsburgh, Penn., June 2–5). IEEE Computer Society, Los Angeles, 1987, pp. 234–243.
21. SHASHA, D., AND SNIR, M. Efficient and correct execution of parallel programs that share memory. *Trans. Program. Lang. Syst. 10,* 2 (Apr. 1988), 282–312.
22. SMITH, A. J. Cache memories. *Comput. Surv. 14* 3 (Sept. 1982), 473–540.
23. SWEAZEY, P., AND SMITH, A. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *13th International Symposium on Computer Architecture* (Tokyo, June 2–5). IEEE Computer Society, Los Angeles, 1986, pp. 414–423.
24. YEN, W., YEN, D., AND FU, K. Data coherence problem in a multicache system. *IEEE Trans. Comput. C-34,* 1 (Jan. 1985), 56–65.

Received August 1989; revised September 1990 and November 1991; accepted November 1991