Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories*

Phillip W. Hutto *Mustaque Ahamad*
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
pwh@cs.gatech.edu, mustaq@cs.gatech.edu

Abstract

We propose and advocate the use of weakly consistent memories in distributed shared memory systems to combat unacceptable network delay and to allow such systems to scale. We survey proposed memory correctness conditions and demonstrate how they are related by a weakness hierarchy. Multiversion and messaging interpretations of memory are introduced as means of systematically exploring the space of possible memories. Slow memory is one such memory that allows the effects of writes to propagate *slowly* through the system, eliminating the need for costly consistency maintenance protocols that limit concurrency. Slow memory possesses a valuable locality property and supports a reduction from traditional atomic memory. Thus slow memory is as expressive as atomic memory. We demonstrate this expressiveness by presenting two exclusion algorithms and a solution to Fischer and Michael's dictionary problem on slow memory.

1 Introduction

The illusion of a global shared memory across distributed processors provides an attractive alternative for programmers but the reality of network latency presents a difficult problem for system architects. Most distributed shared memory (DSM) facilities are implemented using distributed versions of multicache consistency protocols [13, 19, 8, 31] that scale the unit of sharing to a page of memory to increase performance. Other schemes have been suggested to more closely adapt these protocols, developed initially for tightly-coupled multiprocessors, to distributed environments. But all suffer to some degree from this mismatch.

Since DSM facilities are largely implemented by software protocols, another option for improving performance is to *require less restrictive notions of memory consistency*. DSM architects have already observed that the standard criterion, "reads return the most recent write", is not well-defined in a distributed system in which events can only be partially ordered, in which there may be no single, unique "most recent write". This paper reports results of a study of weaker, less restrictive forms of memory for use in DSM environments. We believe that memories with less demanding correctness requirements, *weak memories*, are viable and appropriate for use in DSM systems. The remainder of the paper supports this conclusion and introduces two particularly interesting weak memories, *causal* and *slow*[1] *memory*.

In Section 2 we review a number of arguments supporting the use of weak memories including Lipton and Sandberg's important Latency Theorem. Section 3 surveys proposed correctness conditions and introduces a method for systematically exploring the space of possible correctness conditions. Our main results, in Section 4, define a collection of novel memories, including slow memory. We give centralized and symmetric mutual exclusion algorithms on slow memory in the following section and discuss solutions to the dictionary problem. Finally section 6 shows that slow memory has a valuable locality property and that any computation on atomic memory can be run correctly using slow memory.

2 Why Weak Memories?

Three arguments support weak memories: first, weakening consistency is a natural, appropriate, and common response to latency; second, access to consistent (atomic) memory must be slow in high latency environments; and third, conventional memory often enforces unnecessary ordering constraints.

2.1 A Natural Response to Latency

Two types of situations commonly call for weakening consistency requirements. In the first situation, consistency requirements are often weakened to *make problem specifications more precise* by recognizing acceptable executions that fall outside the bounds of traditional correctness. One of the most well known examples of the first tradition is the dictionary problem ([7]). Fischer and Michael provided a non-serializable replicated database that used a convergent, best-effort form of correctness for special commutative operations. Other non-serializable but acceptable executions are known [6].

In the second situation, consistency is weakened to *compromise problem specifications*, often in an effort to deal with unacceptable limitations such as communication latency. Gray has argued that high latency is un-

*This work funded by NSF grants CCR-8619886 and CCR-8806358.

[1]The effects of writes are allowed to propagate *slowly* through the system to other processors when that can enhance concurrency; thus slow memory is really fast.

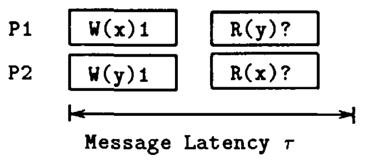

Figure 1: Proof Scenario for the Latency Theorem

avoidable. While today's local networks will behave more like parallel systems with improved communication technology (optical fiber) and higher speed computers, global networks will always suffer the sorts of delays common in current local networks due to the inherent speed of light limitations on signal propagation [9]. The second approach is exemplified by a number of common, practical distributed systems. Sun's NFS provides a weakly consistent version of the UNIX file system over distributed hardware, allowing unintuitive executions since the propagation of updates may be delayed. For example, changing file access permissions (`chmod`) on one workstation may not be immediately reflected in a listing of those permissions (`ls -l`) on a neighboring workstation. Traditional consistency (effects are immediately visible) was apparently too costly for the designers of NFS. Other weakly consistent systems include Sun's Yellow Pages [29], Shard [25], etc.

2.2 Coherent Memory Must Be Slow

In [20] Lipton and Sandberg present a form of weak memory, called PRAM, as part of a *scalable* parallel architecture. They provide a simple argument to demonstrate that there can be no scalable architecture that maintains strong memory coherence. (For now, we use atomic, consistent and coherent interchangeably to refer to intuitive correctness. In section 3 we define a variety of atomic memories.)

Consider a program with two variables, x and y, both initially 0.[2] P1 writes 1 to x and then reads y. P2 writes 1 to y and then reads x. The situation is shown in Figure 1. (Rectangles represent operation intervals. `W(x)1` means that the process noted to the far left in the figure wrote the value 1 to location x.) A consistent memory will not allow the final outcome in which both processes read 0. Assume that the minimum possible time to read plus the minimum possible time to write is less than the communication latency. Then both reads must return 0. The latency is the information delivery time and each processor cannot possibly know of the events that have transpired at the other processor! In Lipton and Sandberg's words: *"No matter how clever or complex a protocol is, if it implements a coherent shared memory, it must be 'slow'. A consistent shared memory cannot be both fast (memory access time independent of τ) and scalable"* [20, p.3].

[2]Variables in examples are always assumed to be initially zero throughout the sequel.

P1	P2
`W(x)1`	`R(x)?`
`W(y)1`	`R(y)?`

x	y
0	0
0	1
1	0
1	1

Figure 2: A Simple Execution on Consistent Memory

P1	P2
`W(x)1`	`R(y)?`
`W(y)1`	`R(x)?`

x	y
0	0
~~0~~	~~1~~
1	0
1	1

Figure 3: Order Implies Consistency Constraints

2.3 Unneeded Ordering Constraints

What are the consequences of abandoning consistent memory? Eggers and Katz [5] report that only 2% of memory requests in common parallel programs involve write contention. It is these requests that would be effected by weak memories. Lipton and Sandberg conclude: *"If this is true then a coherency protocol that significantly delays 98% of all memory references to aid the remaining 2% is of dubious value"*[20, p.4]. Moreover, access to shared data in parallel programs is almost always controlled by higher level synchronization primitives. Providing memory coherency does not eliminate the need for these higher-level forms of synchronization.

Atomic memory implicitly enforces very strong consistency requirements even when not necessary or intended. Consider the four steps in the history (taken from [28]) in Figure 2. All four outcomes shown are possible, depending on the way the four instructions are interleaved. But in the very similar code in Figure 3 the outcome $x = 0, y = 1$ is prohibited: was this intended or did the programmer fail to consider the implicit ordering constraints provided by atomic memory? It would be easy for a programmer to mistakenly order the two reads as in Figure 3, disallowing the (0,1) outcome and imposing an unneeded ordering constraint, when all four outcomes in Figure 2 would be acceptable.

We advocate systems that have wider flexibility and allow programmers to indicate explicitly consistency (ordering) constraints that must be maintained. In other words, we must make explicit in our parallel programming those cases that rely on memory coherence and take explicit measures to enforce that coherence. Being explicit about coherence requires the programmer to be clear about an important area of concurrency (ordering requirements) and may enhance the programming process.

3 How to Weaken Consistency

Memory, commonly regarded, obeys the *register property*: Reads return the value of the most recent write, where reads and writes are described by *operation intervals*. Thus the function of memory is, first, to *order* writes (and reads if necessary), and, second, to store the value of the most recent write for potential reads. The register property is an insufficient specification in environments where "the most recent write" is not well defined.

Proposed correctness conditions for memory form a weakness hierarchy. Reads and writes by a single processor to *private* memory are non-overlapping by assumption. *Serial* executions are, as their name implies, non-overlapping, but allow memory to be shared among multiple readers and writers. Writes to private memory are totally ordered by the program sequence but writes to serial memory are only partially ordered by the several interleaved program orders. Operations issued by distinct processors are related by the interaction with memory itself. Thus, serial memory serves a *communication* function. Both private and serial memory are restrictions of traditional *atomic* memory. Other more permissive definitions are restrictions of *sequentially consistent* (introduced by Lamport in [15]) memory. *PRAM* is the only memory uniformly weaker than sequential consistency proposed to date. We use these three definitions to structure our survey. Finally the weakness hierarchy is revealed using *possibility sets* as a metric.

3.1 Atomic

Memories that allow overlapping, shared access raise the question: which is the "most recent write"? Operations can be ordered by identifying each with a unique point within the interval at which it is said to "take effect". *Static* atomic definitions require operations to take effect for all observers at some specific component event such as read-begin or write-end. Many static definitions are possible depending on the point chosen but the various definitions are not equivalent. Point events are typically not allowed to coincide (but see Lamport [17]). Static atomic definitions characteristically determine a unique register value at all instants.

Dynamic atomic memory generalizes all static definitions by allowing operations to take effect at any point during the operation interval as long as the resulting history is equivalent to some serial execution. During overlapping writes on dynamic atomic memory, readers may *choose* among return values. A sole reader concurrent with two writers may return either of the values being written or the previous value of the register. If two or more readers are concurrent[3] with several writers, choice is still possible but the first value returned constrains further choices. The values chosen must be consistent with reads in some possible serial execution. This choice-making behavior reflects the inherent non-determinism of dynamic atomic memory.

Lamport formalized dynamic atomic memory for a single writer and multiple readers [17] and introduced two weakenings: *regular* and *safe* registers. Both memories behave atomically during serial access but behave weakly during overlapping reads of a write. Regular memory is similar to dynamic atomic memory but two or more concurrent readers need not select values consistent with a serial execution. Reads concurrent with a writer of a safe register may return any possible register value. Lamport defined these notions for the case of one writer but they can be easily generalized. Misra [22] formalizes the multiwriter atomic case. The three definitions form a strict hierarchy: regular is more permissive than atomic, safe more permissive than regular. Herlihy and Wing [10] generalize atomic registers to form *linearizability*, a correctness condition for operations with arbitrary semantics. Linearizability possesses two valuable properties: locality and non-blockingness, not possessed by rivals. We describe locality in a later section.

[3]The reads themselves may be either concurrent or non-overlapping provided they all overlap two concurrent writes.

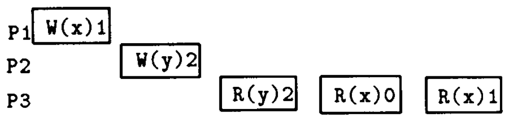

Figure 4: SC Execution with Commuted Op Order

3.2 Sequential Consistency

Sequential consistency is a weak but widely accepted correctness condition introduced by Lamport in 1979 [15]. A multiprocessor is sequentially consistent if: "*... 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*" [15, p.690].

The most striking characteristic of sequentially consistent executions is that the ordering of non-overlapping operations need not be retained by an execution's equivalent serial history. As a result, the effects of operations may appear delayed. If the effects of operations by distinct processors are delayed by differing amounts, operations may actually be commuted in the equivalent serial history. Figure 4 shows a sequentially consistent execution featuring such effects. Sequential consistency requires that each processor's "view" (observation of effects) of other processors must be consistent with the agreed upon equivalent serial execution.

Sequential consistency can be understood as a shared memory counterpart to multiversion serializability [23]. In multiversioning systems (either serializable or sequentially consistent) each write creates a *new version* of the object being written. Writes, therefore, can occur at any time without fear of "overwriting" versions that may still be needed. Questions of correctness reduce to *selecting* the appropriate version for reads to return. The SELECTION protocol then comes to embody and enforce consistency. This is usually done in conjunction with an INVALIDATION protocol that makes certain versions unavailable for future reads. The results of the multiversioning must be equivalent to some single-version, serial interleaving of the program streams. The choice-making of dynamic atomicity is a restricted form of multiversioning. Dynamic atomic memory only allows multiple versions to be introduced by overlapping writes. In contrast, all writes in sequential consistency create new versions.

Lamport's definition of sequential consistency has some startling implications. First, *processors may read values that have not yet been written!* A read that entirely precedes a write may return the (future) value written and remain sequentially consistent! If the two operations P2:W(y)2 and P3:R(y)2 in Figure 4 occurred in reverse order (the read entirely preceded the write), the execution would remain correct (sequentially consistent), even though P3 read a value that had not yet been written. Reading from the future in this way gives rise to non-prefix-closed executions; that is, all prefixes of sequentially consistent executions are not necessarily sequentially consistent. We define *realizable sequential consistency* to be that subset of sequential consistency that contains no executions that "read the future"; that is, only prefix-closed sequentially consistent executions.

Several proposed implementations capture subsets of sequential consistency although no authors appear to have exploited our multiversioning interpretation. Sequential consistency is very popular as a correctness condition in the hardware community and many "advanced" cache consistency algorithms are efforts to capture more sequentially consistent executions. Scheurich and Dubois have proposed one such condition [26]. Brown [3] introduced a more ambitious protocol called *asynchronous multicaching* that buffered invalidations. He extended this technique in a joint work [1] to buffer writes to memory as well as invalidations and called the result *lazy caching*. Even this ambitious effort fails to capture all of the realizable sequentially consistent executions.

3.3 Pipelined RAM

Pipelined RAM (PRAM) is a weak memory proposed by Lipton and Sandberg [20] that "pipelines" writes to memory, allowing the effects of writes (as perceived by other processors) to be arbitrarily delayed. Lipton and Sandberg define PRAM this way: Assume each processor on a reliable network has a complete copy of a global shared memory. Reads and writes are performed on local memory: reads return the local value, writes update the local copy and broadcast the update. Processors receive and process these updates asynchronously and atomically (with respect to local operations). Broadcasts of update values are reliable and point-to-point ordered (all updates from a processor are received in order but the relative order of updates from distinct processors may vary).

PRAM behaves very much like sequentially consistent memory, with both allowing write effects to be arbitrarily delayed. But there is a crucial difference between the two: PRAM allows inconsistent views of the global system state to develop. Sequential consistency disallows such inconsistent views. Consider the correct PRAM history in Figure 5. Inconsistent views may develop if the two updates are "in transit" when the reads are performed. Since the effects of writes may be delayed arbitrarily, they need not "arrive" before the reads complete. After both reads complete P1 believes memory (x,y) to be in the state (1,0) whereas P2 believes memory to be in the state (0,1).

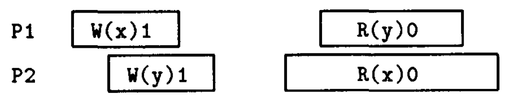

Figure 5: PRAM with Values in Transit

Since the updates will eventually arrive and be applied, continued reads will eventually return the value 1 for both processors and their views will converge. Notice that this example is not sequentially consistent. It is not equivalent to any serial execution of the four statements.

The distinction between PRAM and sequential consistency is subtle and can be expressed formally by differing orders of quantification [11]. Roughly, sequentially consistent processors must all agree on an equivalent serial execution while PRAM processors may individually select equivalent serial histories and these equivalent serial histories may differ from processor to processor.

We have seen one case in Figure 5 where the update values are delayed in transit causing a temporary discrepancy between processor views. In this case the views of both processors eventually converge. They agree on the contents of memory but disagree on the order in which operations were applied to bring memory to its final state. Consider concurrent writes to the same location. P1 writes 0 and P2 writes 1. On subsequent reads P1 returns 1 and P2 returns 0. They have observed the writes to occur in differing orders, as above, but in this case the processors may fail to converge, differing on their view of memory for the remainder of the execution. While PRAM processors may disagree on the order of writes by distinct processors and, ultimately, on the contents of memory, processor views may not diverge arbitrarily. All processors must mutually agree on the order of observed effects of individual processors.[4] Finally, note that PRAM executions in which processors fail to communicate are degenerately correct. We must assume that writes of memory are eventually effective, that they are eventually applied at all processors. This is not unusual—atomic memories have strong implicit liveness conditions. They all assume that reads and writes of memory eventually terminate and that write effects are available immediately upon termination.

3.4 The Memory Hierarchy

We can visually and qualitatively relate the memories we have surveyed. A *concurrency set* as used by database theorists [23] contains all executions allowed by a correctness condition or scheduler (implementation). Various conditions can be partially ordered by set containment. Using Venn-style diagrams, weaker conditions have greater "area". Thus view serializability is a weaker (more inclusive, less restrictive) condition than conflict serializability. We define a *possibility set* similarly as an execution set for memories, but there is a crucial difference.

[4]Since all writes need not be observed (read) by all processors, processors need only agree on the order of writes (effects) actually read (observed).

Database histories are execution *schema*, showing interleaved operations, but without values for reads and writes. These schema represent useful information because serializabilities actually disallow various interleavings. A weaker condition allows more interleavings, hence, more concurrency. For memories, there are *no* disallowed interleavings! There is always *some* value that can be returned. (That is, memories are typically *non-blocking*.) Memory histories are *valued* executions. The executions contained in possibility sets show, unlike concurrency sets, the number of *different values* that a read may return. Conditions with more possibilities allow more concurrency by providing schedulers with greater flexibility.

Reviewing the various atomic memories, we find that *private* histories are contained in the possibility set of *serial* histories (each private history is a serial history with a single active processor). Similarly serial histories are a subset of any of the various forms of static atomic memories. The various definitions of static atomic memories are inequivalent but all are less permissive than dynamic atomic memory. In fact dynamic atomic is the union of all possible static atomic memories; that is, dynamic atomic memory allows all executions allowed by any possible static atomic memory. Dynamic atomic memory is also linearizable. Linearizability (LIN) is equivalent to dynamic atomicity when restricted to objects with read and write operations. (*private* $\subset$ *serial* $\subset$ *statics* $\subset$ *dynamic (linearizable)*)

Moving out beyond dynamic atomic we find two separate hierarchies. Lamport's safe (SAFE) and regular (REG) memories are weakenings of (dynamic) atomic (DA) memory that introduced additional possibilities during periods of concurrency but behaved atomically otherwise. ($DA \subset REG \subset SAFE$) The second hierarchy is more complex. The realizable sequentially consistent (RSC) executions are considerably more permissive than dynamic atomicity. Scheurich and Dubois' condition (SD), Brown's asynchronous multicaching (AM), and Afek, Brown, and Merritt's lazy caching (LC) are all attempts to capture larger sets of the RSC executions. Finally, PRAM is even more lenient than sequential consistency. ($DA \subset SD \subset AM \subset LC \subset RSC \subset PRAM$)

How are the two hierarchies extending out beyond dynamic atomic related? Regular and safe are very permissive during concurrent access. Sequential consistency is oblivious to concurrency but allows considerable leniency throughout the computation. Regular and safe are incomparable to realizable sequential consistency. While all three conditions share among them the dynamic atomic executions, there remain safe (regular) executions that are not RSC. Similarly, there are RSC executions that are neither safe nor regular. This complicated situation is shown by the solid line bounded regions in Figure 6.

Recall that regular memory allows any value being written to be returned during overlapping access. We can define a new memory called *weak* that requires only that reads return some value previously written. Weak memory contains both PRAM and regular but is incomparable to safe. We can contain both safe and weak memory by a similar move. Define a memory that simply returns some allowed value of the register under all conditions. Such a memory is effectively a random number generator so we call it *random* (RAND). Random contains all forms of memory we have encountered. Random memory is too weak to support interprocess communication but serves as a reasonable "background set" for all possible definitions of effective memory. The final hierarchy is shown by the addition of dashed and dotted regions in Figure 6.

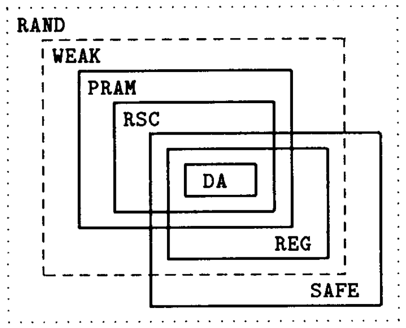

Figure 6: Possibility Sets Relating the Two Hierarchies

4 Varieties of Memory

There exists a rich space of possible memories.

4.1 Causal Memory

Recall that sequential consistency requires all processors to agree on the order of observed effects. In contrast, PRAM requires only agreement on the order of observed effects of individual processors. Events by distinct processors may be observed in differing orders. There is an intermediate possibility. In [14] Lamport introduces the notion of *potential causality* to capture the flow of information in distributed systems. We may apply potential causality to a distributed shared memory by relating message-send to write and message-receive to read. Events (reads and writes) initiated by a single processor are totally ordered by potential causality. Reads of remote writes are also related by potential causality (`P1:W(x)1` $\rightarrow$ `P2:R(x)1`).

Define *causal memory* that requires all processors to agree on the order of causally related effects (writes) but allows events not related by potential causality (*concurrent* events) to be observed in differing orders. Figure 7 gives a correct execution on causal memory. (All operations are on the same location x for simplicity.) `W(x)1` and `W(x)2` are potentially causally related (`W(x)1` $\rightarrow$ `W(x)2`) since P2 reads P1's write (information flows from P1 to P2). Consequently, reads of all processors must respect this ordering. `W(x)2` and `W(x)3`, on the other hand, are unrelated by potential causality. Processors (such as P3 and P4) may observe these writes to have occurred in differing orders. Causal memory is an analog of the ISIS *causal broadcast* [2] and is supported by the same arguments used to advocate causal broadcast. In fact, Schmuck [27] proves that a very large class of problems captured by his notion of a *formal specification* all have

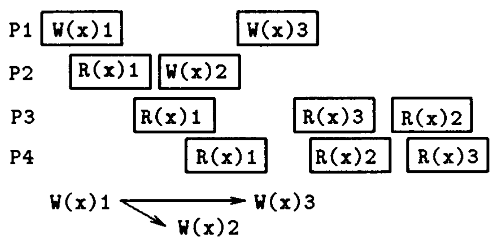

Figure 7: Causal Memory

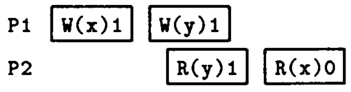

Figure 8: Slow Memory: Writes to x, y Commute

CBCAST implementations. This result transfers to casual memory and thus proves that all formal specifications have a causal memory implementation as well. Note that dynamic atomic memory allows concurrent writes to be observed in either order but all observers must agree. Causal memory allows concurrent writes to be observed in either order without agreement among observers.

4.2 Slow Memory

PRAM is a *processor relative* weakening of memory: all processors must agree on the order of all observed writes by a single processor. A *location relative* weakening of memory would require only that all processors agree on the order of observed writes *to each location* by a single processor. Define *slow memory* as follows: reads must return *some* value that has been previously written to the location being read. Once a value has been read, no earlier writes to that location (by the processor that wrote the value read earlier) can be returned. Local writes must be immediately visible. Slow memory is a location relative weakening of memory.

All PRAM executions are allowed by slow memory but there are correct executions on slow memory not allowed by PRAM. All memories that we have considered so far (except safe and regular) guarantee the sequencing of effects by a single processor. That is, if a processor writes a value to x and then to y, observing the value written to y ensures that the value of x is either the value previously written or some value written later. Slow memory decouples writes to distinct locations, even if they are by the same processor. Figure 8 shows a correct slow execution.

Slow memory is a very weak memory; it may seem unusably weak. We contend that causal memory, slow memory and other varieties of weak memory are no more difficult to program or use than traditional memories once the basic interprocess communication "inference rules" are grasped. In the following sections we present solutions of non-trivial problems on slow memory and show a universal reduction between slow and atomic memory.

Once memory is conceived as a multiversioning read-write system, controlled by UPDATE, SELECT, and INVALIDATE protocols, new memories emerge by systematic variation of these protocols. Particularly important are the *invalidation rules* used to disallow old versions. In this model, reads often fix the order of writes by invalidating older versions once a value is read. Thus different memories have differing read invalidation rules. A version (effect) that has been invalidated is *ineffective*. In the full paper [12] we present a systematic table of possible invalidation rules of the form: "Once a write is observed, earlier writes to X by Y are ineffective for Z". By casting the invalidation rules in a common formalism it is easy to contrast two memories such as PRAM and slow memory. Moreover it provides a systematic means of exploring the space of possible memories, allowing us to identify new, novel memories.

```
Peterson's
Entry Protocol:
$x_i$ := T;
$y$ := $j$;
wait $\neg x_j \vee y = i$;
```

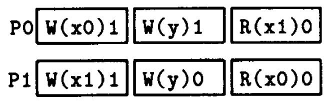

Figure 9: Traditional Exclusion Fails on Slow Memory

5 Programming with Slow Memory

Slow memory can be expressive and efficient when programmed appropriately. In this section we show solutions to the exclusion problem[5] and to the dictionary problem to demonstrate this expressiveness. Special programming *idioms* for slow memory are discussed as a means of simplifying common programming tasks.

5.1 Exclusion

Traditional mutual exclusion solutions (such as Peterson's [24]) fail on slow memory. The entry protocol for Peterson's two-process algorithm is shown in Figure 9 with a partial execution. When x_i is true, process i is *trying* to gain entry to the critical section. The variable y arbitrates *turns* when both processes seek entry to the critical section. Slow memory allows the read of x_i by each process to return the initial value, 0, (see the "values in transit" example for PRAM in Figure 5) thus providing non-mutually exclusive access to the critical section. Peterson's algorithm relies on the strong coherency guaranteed by atomic memory.

Exclusion *is* possible, however, on slow memory. Figure 10 shows a simple n-process algorithm expressed in Dijkstra's guarded command notation [4]. The notation "**wait** e" is shorthand for "[$\neg e \rightarrow$ *skip*]". The algorithm requires two slow bits, request and acknowledge, per process. The server scans request bits until one is found true. That client is granted the critical section and the server waits for release. A fair scan (easily implemented

[5]Lamport rules out as solutions to the mutual exclusion problem "processes that take turns entering, or declining to enter, their critical sections" [16] saying that such algorithms solve the producer/consumer problem. Our algorithms are of this character so we refer to them as *exclusion* algorithms.

```
Client $i$::[
  T $\longrightarrow$                      Mutex Server:: *[
    NCS;                                $req_i$ $\longrightarrow$
    $req_i$ := T;                          $req_i$ := F;
    **wait** $ack_i$;                      $ack_i$ := T;
      CS;                                  **wait** $\neg ack_i$;
    $ack_i$ := F;                       ]
]
```

Figure 10: Centralized, n-Process Exclusion on Slow

by the server resuming at process $i+1$ after granting mutex to process i) ensures lockout-free execution. Reading "old" versions (the problem with Peterson's algorithm) is avoided by "handshaking" reads and writes. Since slow memory preserves overwrite semantics (writes can not be "duplicated"), once a value is read, previous values (written by the same process) are unavailable for reading. Sequencing of writes is also important. For example, if the write of F to req_i by the server is done *after* the **wait** statement, a subsequent request by process i may be lost, blocking that process indefinitely.

More complex algorithms are possible. In the full paper [12] we present a symmetric two-process exclusion algorithm on slow memory that augments the client processes of the n-process algorithm with symmetric server processors. In effect, the two additional processes cooperate to replace the central mutex server. Our slow memory exclusion algorithms have an interesting property. Mutual exclusion is usually sought to guarantee consistency of sensitive data. The critical section "guards" the consistency of data modified within it. While our algorithms guarantee that no two processes will be granted simultaneous access to the critical section, they do not provide *logical* mutual exclusion, they do not ensure the consistency of data modified within the critical section. Consider two processes, granted consecutive access to the critical section, that each write a and then b. Slow memory allows a third process, subsequently granted exclusive access, to read the value written by P1 to a and by P2 to b, as if the two processes had simultaneous access to the critical section! Our exclusion algorithms provide *physical* but not *logical* exclusion. Mutual exclusion on slow memory does not provide implicit *view atomicity*, as it does on atomic memory. We must use explicit techniques to achieve view atomicity on slow memory.

5.2 Programming Idioms

Note that changing the *granularity* of reads and writes is essentially a question of view atomicity. In the example in the previous section, reading and writing a and b as an *aggregate* or record would provide logical exclusion. Cooperating processes can provide view atomicity for arbitrarily large updates on slow memory at the expense of extra storage. Thus processes can effectively vary their read/write granularity. An aggregate write could be programmed on atomic memory by signaling to the reader that the write has completed. But slow memory does not guarantee that the aggregate bits will be visible before the signal bit; they are distinct locations, although written by the same process. A simple coding trick provides the solution. If the reader expects $\lfloor n/2 \rfloor$ out of n bits to be set by each aggregate write, then $m = \binom{n}{n/2}$ distinct values can be transmitted. Once all m bits are received, the reader can set all bits high in acknowledgement and to prepare for the next aggregate write.[6] Since this facility can be implemented using the "native" power of slow memory, addition of a special, efficient primitive to achieve the same purpose does not alter the possibility set of slow memory. Message systems (even unreliable ones) typically provide the capability of transmitting arbitrarily large messages "atomically". Providing view atomicity to an aggregate update is an example of a *programming idiom*: a conventional technique, applied in a variety of circumstances. We believe identifying a catalog of such idioms is the key to programming weak memories.

We call another valuable idiom the *invalidating write*, named after the invalidation process in cache consistency protocols. The invalidating writes provides a means for synchronizing all processes with respect to accesses of a single location. Such a write guarantees that no process will read values written earlier than the invalidating write. It effectively invalidates all previous versions of the location present in the system, making them inaccessible for future reads. Surprisingly, even this strong primitive can be implemented natively.

5.3 The Dictionary Problem

The *dictionary problem* [7] is to implement an association table in an unreliable, asynchronous environment, satisfying four constraints: consistency, efficiency, fault-tolerance, and convergence. We have developed a slow memory solution to the dictionary problem that "distributes" the dictionary and reduces latency costs by allowing dictionary views to diverge. This solution demonstrates slow memory programming idioms and reveals interesting relationships between weak memories and fault-tolerance properties. Due to space limitations we refer the reader to the full paper [12] for a description and discussion of this algorithm. We note, however, that Lanin and Shasha [18] have published a wait-free set manipulation algorithm that is surprisingly similar to our slow memory dictionary, suggesting that slow registers may be higher in the wait-free hierarchy than traditional atomic registers.

6 Properties of Slow Memory

Slow memory possesses valuable locality and reduction properties. Property p is *local* if the local correctness of all participants implies the global correctness of the system. Herlihy and Wing argue for locality because: *"... objects can be implemented, verified, and executed independently."* Slow memory is local because the value returned by a read only depends on the history of operations on that same location. In contrast, PRAM is not local. Figure 11 shows an incorrect PRAM execution

[6]Slow memory programming remarkably parallels the theory of *delay-insensitive (DI) codes*[30]. Such codes are used to implement *self-timed circuits*[21, ch. 7]. Our signaling trick implements a *Sperner code*, the optimal DI code.

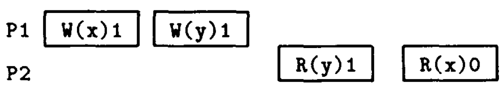

Figure 11: PRAM is Not Local

affecting objects x and y in which the local object histories *are* correct. We can also reduce any execution on atomic memory to an equivalent execution on slow memory. The central mutex server of our n-process exclusion algorithm becomes a *step* server, that non-deterministically grants a step to some process. That process uses the "handshaking" technique to transmit the type of operation it wishes to perform, the location it needs to read or write, and the value written (if necessary). The step server maintains the "shared" memory and each read of the shared memory becomes a read of a value written by the server. Since slow memory requires writes to a single location by a single process to be ordered, our system behaves serially (hence atomically). Thus any program that runs correctly on atomic memory can be transformed (algorithmically) into an equivalent program that runs on slow memory and produces the same results.

7 Conclusions

We believe that weak memories offer a novel and fruitful area of study. We plan to continue our work with increased attention to practical programming and implementation issues.

References

[1] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. In *Symp. on Parallel Alg. and Arch*, pages 209–223, June 1989.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), Feb. 1987.

[3] G. M. Brown. Asynchronous multicaches. Technical report, Cornell Univ., School of EE, 1988. (Submitted to *Dist. Comput.*).

[4] E. W. Dijkstra. Guarded commands, nondeterminancy and the formal derivation of programs. *Commun. ACM*, 18:453–457, Aug. 1975.

[5] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluations. In *Int. Symp. on Comp. Arch.*, pages 373–383, 1988.

[6] A. A. Farrag and M. T. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Trans. Database Syst.*, 14(4):503–525, Dec. 1989.

[7] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Prin. of Database Syst.*, pages 70–75, 1982.

[8] B. D. Fleisch. Distributed shared memory in a loosely coupled distributed system. In *Spring COMPCON '88*, 1988.

[9] J. N. Gray. The cost of messages. In *Prin. of Dist. Comput.*, Aug. 1988. (1987 keynote address).

[10] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. Technical report, CMU, Dept. of CS, Dec. 1987. (Originally in *Symp. on Prin. of Prog. Lang.*, Jan. 1987).

[11] P. W. Hutto. *Weakly Consistent Memories.* PhD thesis, Georgia Tech., School of ICS, 1990. *In progress.*

[12] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. Technical Report GIT-ICS-89/39, Georgia Tech, School of ICS, Oct. 1989.

[13] Y. A. Khalidi. Hardware support for distributed object-based systems. Technical Report GIT-ICS-89/19, Georgia Tech., School of ICS, June 1989. (PhD Thesis).

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C–28(9):690–691, Sept. 1979.

[16] L. Lamport. The mutual exclusion problem: Part I - A theory of interprocess communication. *J. ACM*, 33(2):313–326, Apr. 1986.

[17] L. Lamport. On interprocess communication; Part I: Basic formalism. *Dist. Comput.*, 1(2):77–85, 1986.

[18] V. Lanin and D. Shasha. Wait-free concurrent set manipulation. In *Prin. of Database Syst.*, 1988.

[19] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[20] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., Dept. of CS, Sept. 1988.

[21] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[22] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Prog. Lang. Syst.*, 8(1):1420–153, Jan. 1986.

[23] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[24] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, June 1981.

[25] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Trans. Softw. Eng.*, SE-13(1):39–47, Jan. 1987.

[26] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Int. Symp. on Comp. Arch.*, pages 234-243, June 1987.

[27] F. B. Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* PhD thesis, Cornell Univ., Dept. of CS, Aug. 1988.

[28] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Prog. Lang. Syst.*, 10(2):282–312, Apr. 1988.

[29] *Sun Network File System (NFS) Reference Manual.* Sun Micro., Inc. , Mountain View, CA, 1987.

[30] T. Verhoeff. Delay-insensitive codes - An overview. *Dist. Comput.*, 3:1–8, 1988.

[31] L. Wittie and C. Maples. MERLIN: Massively parallel heterogeneous computing. In *Int. Conf. on Parallel Proc.*, 1989.