A Cache Technique for Synchronization Variables in Highly Parallel, Shared Memory Systems

by

Wayne Berke

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #151
December, 1988

This work was supported in part by the Applied Mathematical Sciences Program of the US Department of Energy under contract DE-AC02-76ER03077 and in part by the National Science Foundation under grants DCR-8413359, and in part by I.B.M. under joint study agreement N00039-84-R-0605(Q).

ABSTRACT

Caches have traditionally been used to lower the average latency of memory access. When paired with the individual CPUs of a multiprocessor, they have the additional benefit of reducing the overall load on the processor-memory interconnection. Since synchronization variables have been identified as centers of memory contention, we have looked at methods of utilizing the cache to minimize this effect. A technique of ‘‘polling the cache’’ is proposed to deal with this problem. Consistency within the caches is maintained with a bottleneck-free update facility that exploits the topology of the multistage network. Since an indiscriminate broadcast-on-write policy can lead to severe network congestion at high levels of parallelism, we selectively invoke these updates from the software. We illustrate our methods with a number of useful synchronization algorithms and present simulation results that support the feasibility of our design. In addition to providing support for basic synchronization operations, our methodology is generally applicable to all parallel algorithms that utilize polling.

1. Introduction

The shared memory MIMD computer is becoming the vehicle of choice for many parallel programmers due to the simplicity of the programming model when compared to that of distributed memory machines such as hypercubes and multi-dimensional meshes. Although the shared memory paradigm can be imposed on these latter machines,[1] the dichotomy between software model and underlying architecture will limit peak performance.

A dominant characteristic of any shared memory machine is the manner in which PEs are connected with memories. Those with a small to moderate number of PEs can work well with either a crossbar [WuBe72] or a simple bus [ThGF88] [Hill86]. However, in considering a highly parallel machine (with hundreds of PEs, for example), busses become too easily saturated and crossbars become too expensive. In constructing an N-PE multiprocessor where N is large, a multistage network [GoLi73] [Pate81] [Gott87] can offer a good compromise between hardware cost ($\Theta(N \log N)$) and bandwidth ($\Theta(N)$).[2]

Unfortunately, multistage networks are also subject to saturation especially while supporting programs that display non-uniformity in their access patterns [PfNo85]. When such *hot spot* behavior occurs, the memory contention becomes so severe that it affects the latency of requests to all of memory. Pfister and Norton identify synchronization variables as a prime source of such hot spots—a finding that is supported by other studies [LiAb85] [DaHe88] as well. To deal with this adverse behavior, they advocate the method of identifying like operations directed to the same memory location and *combining* these operations within the network. This can potentially result in N requests being transformed into one (see [Gott87] for a detailed description). However, the cost of a hardware based combining facility may be considered prohibitive.

We propose an alternative means of reducing network contention by extending the conventional role of the private cache. Our method is motivated by the observation that a number of synchronization algorithms dedicate a substantial amount of time *polling* (*i.e.*, periodically checking the contents of an agreed-upon memory location). In

---

[1]A hypercube could treat all the local memories as a global address space, for example. A non-local reference would then be transformed into a message that requested the contents of some remote memory word.

[2]Asymptotic bandwidth analysis is difficult for multistage networks with *buffered* switch nodes. Analytic methods have shown that unbuffered networks have bandwidth $\Theta\left(\frac{N}{\log N}\right)$ assuming random requests [KrSn83]. Although there are no analogous results for buffered networks, simulations indicate that buffering results in bandwidth that is linear in N.

a network based multiprocessor, this can result in a large portion of the network traffic for the entire synchronization operation. We consider this additional load on the network to be unnecessary. By allowing synchronization variables into the cache and ‘‘polling the cache,’’ we can effectively eliminate this overhead.

We consider a programming model that distinguishes *private* (each process maintains a unique copy) from *shared* variables. Assuming that transient memory effects unrelated to parallelism (due to paging and swapping, for example) are handled in software, all private variables and shared read-only variables may be safely kept in any processor cache. Shared read-write data, such as synchronization variables, have to be treated specially however. In particular, we must ensure that the cache representatives of each shared variable accurately reflect their values in main memory. This is commonly referred to as the multicache *consistency* or *coherency problem*.

There have been several attempts to address the consistency problem in the general case. The hardware based methods [CeFe78] [ArBa84] [ASHH88] use directories with global state information for each sharable item that appears in any cache. The usual convention is that data can be in one of two states: *shared read-only* or *exclusive read-write*. Before any PE can write a shared-cacheable datum, it must initiate an *ownership* protocol with the directory to ensure that it has an exclusive copy in its cache. This potentially entails invalidating or updating individual lines in other caches. Although these operations can occur in a decentralized manner in bus based machines, they induce considerable performance overhead in multistage networks. In particular, when these methods are applied to *volatile* (*i.e.*, heavily shared and frequently written) data, the resulting overhead from consistency operations can seriously impair overall performance.

The software methods [ChVe88] [CyKM88] use traditional data flow techniques to statically determine when variables can be safely cached. However, these often result in the caching only the least volatile data, which can have a limited effect on reducing network contention. A major reason for this is that the unpredictability of data access patterns imposes a ‘‘worst case’’ approach to enforcing consistency.

In light of this, our method extends cacheability to a very restricted class of shared read-write variables—synchronization variables. These variables are usually dismissed from such consideration because of their high degree of volatility. However, we take advantage of the fact that they appear in only a small number of program modules (usually one or two). The decreased number of possible access conflicts allows for a more selective use of consistency operations. Caches with stale values are made consistent by the UpdateAll operation (described below), which is issued under software control.

Our methods are sufficiently general to be applicable to any multiprocessor configured with a *banyan* network [GoLi73]. Note that *delta* [Pate81] and *omega* networks [Lawr75] are subsets of the banyan class. These networks are all characterized by the property that a set of paths exist from each memory node to all the processors. Furthermore, these paths comprise a tree that fans out through the various network stages. Our proposed design draws upon this property to implement consistency operations networks that do not induce serial bottlenecks.

There is little in the way of extra hardware required; in particular, there is no need for additional tag bits in each cache entry or for directory structures at the network-memory interface. Although we require the caches to support write-through operations, caches with hybrid policies [Mcau86] can be used to retain the performance advantages of copy-back. We assume that our caches are physically tagged with a one word (four byte) line size. The implications of relaxing these assumptions are investigated in section 8.

This paper is organized in the following way. In section 2, we review the differences between synchronization methods that utilize polling and those that rely on interrupts. Section 3 outlines the consistency problem, how it is dealt with in bus based systems, and proposes a network based adaptation of these techniques. We introduce the UpdateAll operation in section 4 and illustrate its use within several key synchronization algorithms in section 5. In section 6, we present simulation results that support the feasibility of our methods and suggest the kinds of applications that can benefit from them. Section 7 discusses other possible applications of our technique, while section 8 describes issues related to a possible hardware implementation.

2. Busy-waiting vs. block/wakeup synchronization

In general, there are two means of implementing synchronization operations:

block/wakeup	suspending execution and relying on an asynchronous interrupt to resume execution.
busy-waiting	polling an agreed-upon memory location until it contains a certain value.

The advantages of the first method include the efficient utilization of two critical resources: processors and network bandwidth. In a situation where communication is infrequent and/or unsystematic, it is preferable to suspend a process while it is waiting for a message. The PE it had been executing on can then be utilized in a more productive fashion.

Polling, on the other hand, provides a near-instantaneous response that is impossible in the other method, which requires the additional overhead of a context switch. Even if interrupts are implemented completely in hardware, the processor must read in some amount of process state before it can actually react to a message. The price of this good response is to tie up processor and network resources.

A number of hybrid schemes attempt to retain the benefits of both methods. Polling that is tempered by an exponential backoff, where memory accesses become less frequent over time, can be effective in certain applications. After a long period of unsuccessful polling, the process may even be suspended entirely. This softens the impact on the network while allowing for reasonable response times in the average case. Unfortunately, these two characteristics are at odds with each other. The more critical the window of response, the more frequently we must poll across the network.

Our technique is motivated by an attempt to merge these contending constraints. By allowing PEs to poll copies of the synchronization variables within their caches, we combine the low response time of block/wakeup synchronization with the reduced network traffic of the busy-waiting method. When a PE wants to signal the others, it first modifies the synchronization variable in memory and subsequently communicates this update to all the caches. Note the similarity here with Sequent's parallel spin-locks [ThGF88] [LoTh88] and with the test&test&set primitive of [RuSe84]. These latter methods are only applicable to bus based systems, however, whereas our technique is more generally suited to large scale parallel systems. In the following two sections, we present our methods for implementing this technique.

3. The consistency problem

In order for ‘‘polling the cache’’ to be feasible, cache lines corresponding to shared data must be kept consistent with their globally accessible values in memory. The ‘‘classical’’ solution (as it is referred to in [CeFe78] [ArBa84]) accomplishes this by following every modification of a shared variable with an update or invalidation of all its copies in any local cache. A bus based architecture with *snooping caches* allows this to be done in a decentralized manner by having each cache monitor the bus [Good83] [RuSe84] [LoTh88]. It can then detect when another PE modifies a shared variable for which it has a valid cache entry. In this way, stale entries may be invalidated. Even multiprocessors configured with multiple, hierarchically organized busses can maintain consistency among the different levels in an analogous manner [Wils87].

This kind of passive monitoring is not possible in a parallel computer with a multistage interconnection network—there is simply no central resource to monitor. Furthermore, the caches are too distant from the data to systematically detect inconsistencies. Without a global monitor, we must rely on the PE that actually modifies a shared datum to alert the other PEs that their local versions are out of date.

A direct application of the classical solution would thus accompany all modifications to any shared variable *S* with a directive to invalidate or update any cache lines that contained values for *S*. This could be implemented in a multistage network by providing a *broadcast* mechanism whereby each PE's cache would constitute a leaf of the tree rooted in *S*'s memory module. Figure 1 illustrates how this is done in an omega network, where any of the 2×2 switches that receives the broadcast signal simply propagates it to both output ports on the return path. Note that this mechanism avoids the serial bottleneck that occurs in *cross-invalidate* methods [CeFe78] [ASHH88] where processor by processor notification is required.

If we systematically issue a broadcast-update or broadcast-invalidate[3] with each store into shared memory, a

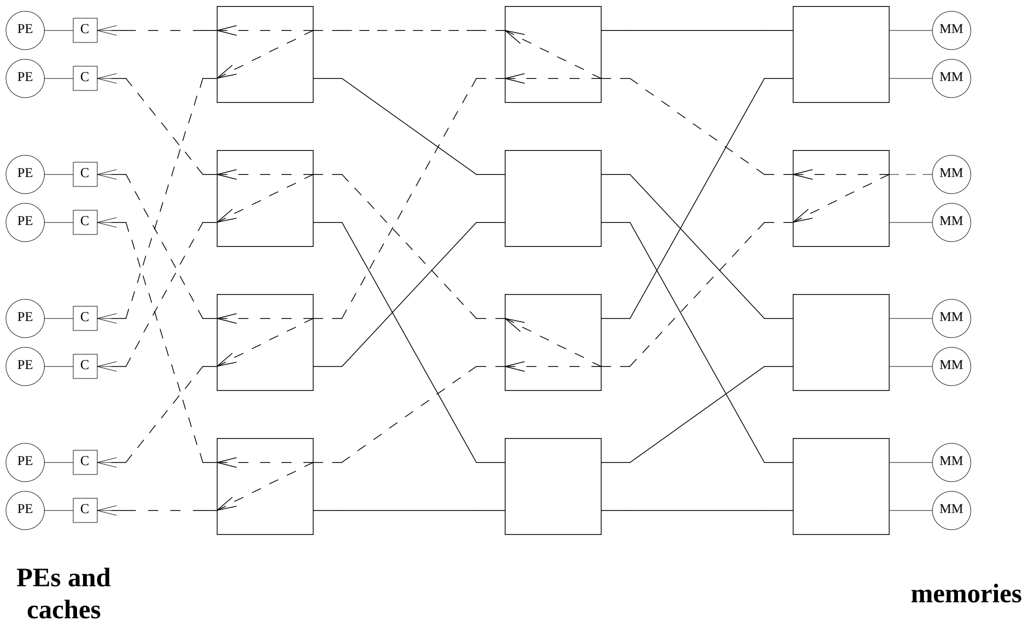

Figure 1. Broadcasting through an omega network

[3]Note that updating may be superior to invalidation because it relieves each PE from having to issue a subsequent network request.

coherent caching policy results. However, this puts significant bandwidth demands upon the network. Each time shared variables are written, we must send along an accompanying broadcast-update. Note that the latter operation involves $\Theta(N)$ switches as opposed to $\Theta(\log N)$ for an ordinary memory access. If shared writes constitute a significant portion of our instruction mix, unrestricted updates lead to network behavior in which the queuing delays for the return path (from the memories to the PEs) become excessive.

Our method limits the additional network load in two ways: first, we only extend caching to selected synchronization variables; second, we use a variant of the broadcast-update operation that is under software control. These two restrictions allow us to tailor our algorithms in such a way as to minimize the number of broadcast operations. As will be shown, this turns out to be fairly simple in many cases. Finally, by permitting loads from synchronization variables to be satisfied by the cache, algorithms that ensure a high load-to-store ratio with respect to these variables can substantially reduce hot spot traffic.

4. The UpdateAll operation

We propose a generalized variant of the broadcast operation where

UpdateAll *v* to *e*

results in:

(1) evaluating the expression *e*

(2) issuing a network request that results in broadcasting the pair of values *v*,*e* to all the processor caches.

(3) when each PE's cache receives this broadcast, it determines whether or not it holds a valid entry for *v*; if so, it updates the entry to contain the value *e*.

Thus, an UpdateAll operation only affects the contents of the processor caches; it has no effect on main memory. The question arises of whether an UpdateAll for some variable interferes with a pending load of the same variable.[4] Since a load only generates a network request after a cache miss and an UpdateAll only modifies caches with a valid entry for the target variable, the two operations can not affect the contents of the same cache. Other kinds of network requests may present potential conflicts, however, but we postpone discussion of this issue until the next section.

Note that although we will generally use UpdateAll to broadcast the latest determinable value in memory, this is not a strict requirement. In fact, section 5.1.2 provides an example where it is expedient to use an update value that may be different from the real value in memory.

In the following section, we present examples of synchronization algorithms that utilize UpdateAll. The algorithms all contain sections in which the processor enters a busy-waiting loop that polls the synchronization variable. After its first read, the variable will be brought into the cache. Thus, if the poll is unsuccessful, the loop will continue and subsequent reads will be satisfied from the cache. The most common example of this kind of polling would thus appear in a simple loop:

[4] In the following discussion, we do not directly address the issue of pipelining requests through the network. For simplicity, we assume that each PE is forbidden from issuing more than one outstanding request to the same memory location.

```
while (S ¬= goodval)
   continue
endwhile
```

The polling process will then not exit the loop until some other process issues the pair of instructions:

```
S = goodval
UpdateAll S to goodval
```

Note that decoupling the update to memory from the cache update implies a temporary state of inconsistency. In the above, however, if a polling process accesses stale data from the cache, it always does the conservative thing (continues to poll). Thus, the temporary inconsistency will not affect the overall correctness of algorithm.

5. Synchronization Algorithms using UpdateAll

We present algorithms implementing three commonly encountered synchronization operations: binary semaphore, counting semaphore, and barrier synchronization. The algorithms are based upon standard models found in the literature on operating systems and parallel processing [Shaw88] [GoLR83]. Each algorithm has an associated synchronization variable and a synchronization primitive. The former can be a semaphore, barrier counter, etc. The latter is a low level instruction supported by the target hardware, notably *test&set* or *fetch&add*. For simplicity, we adopt the convention that all shared variables are non-cacheable unless explicitly indicated with a cacheable declaration.

Each cache must be able to recognize an UpdateAll directive coming from the network. Other than this, cacheable loads and stores are handled in the conventional fashion. However, since the aforementioned synchronization primitives are not generally applied to cacheable data, we should define the semantics when they are. These primitives are really composite read-modify-write instructions which require hardware locks at the memory to enforce atomicity. When either primitive is directed at a cacheable synchronization variable, the following transpires: (a) the PE issues a network request for the variable irrespective of whether it was in the cache or not; (b) the main memory representative of the variable is atomically read out and modified according to the semantics of the primitive, and; (c) the value fetched in (b) is returned to the PE and entered into the cache.

Previously, we argued that network loads for any particular variable can never conflict with UpdateAll for the same variable since the two operations affect disjoint sets of PEs. When a PE executes a synchronization primitive however, it must issue a network request for load even when the target variable is cache-resident. This raises the possibility that an UpdateAll operation may arrive at the cache of a processor while it is awaiting the response to the synchronization primitive. In this case, the latter operation will overwrite the cache entry and the effect of the UpdateAll may not be seen by the processor. However, since we adhere to the convention of modifying the variable in main memory before issuing the UpdateAll, the PE is returned the most recent version in either case.

Another possible conflict occurs when UpdateAlls to the same variable are issued by different PEs. Again, the second operation to arrive will overwrite the modifications made by the first. For this reason, we design the algorithms so that either: (1) concurrent UpdateAlls to the same variable are prevented from occurring, or (2) when such concurrent UpdateAlls are allowed, we always update to the same value. Note that these restrictions comprise a well known variant of the CRCW computing model. In sections 5.1.1 and 5.1.3, we present algorithms that ensure (1) whereas the algorithm in 5.1.2 ensures (2).

Since the enhanced algorithms are based upon models that are assumed to be functionally correct, we discuss the effect of our enhancements (*i.e.*, polling/updating within the cache) upon the semantics of the original routines.

As mentioned earlier, temporary cache inconsistencies result in a conservative state where processes remain in the polling loop. In light of this, we examine the possibility that the new semantics induce an indefinite postponement within the polling loops that could possibly lead to deadlock.

5.1. Semaphore algorithms

The implementation of both binary and counting semaphores utilize the test-modify-retest method [GoLR83]:

(1) The invoking process polls the synchronization variable (semaphore) in a tight loop **(test)**.

(2) After (1) is successful, the synchronization operation is invoked upon the semaphore **(modify)**.

(3) The return value from (2) is checked and if it indicates an failure to obtain the semaphore, the program branches back to (1) **(retest)**.

The only functional differences between the original algorithms and our enhanced versions are:

- the fact that processes engaged in (1) are polling variables from their caches.
- when a process does a V, that procedure includes an UpdateAll request which results in allowing all polling processes to break out of their loops.

The original versions of these algorithms are known to guarantee mutual exclusion. Furthermore, since we have not modified the **modify** or **retest** portion of the algorithm in any way, it is reasonable to believe that our enhanced algorithms retain these properties provided the cache polling in **test** does not lead to deadlock. We will show that any process within the polling loop must be able to break out it within a finite amount of time. Note that we can ignore the issue of cache evictions since an eviction forces the routine to reload the variable from memory, thereby yielding the behavior of the original algorithm.

5.1.1. Binary semaphore

The first example is the binary semaphore algorithm illustrated in Figure 2. Assume that these semaphore

---

```
P(b)   { initially b = FALSE }
cacheable b
loop:  while b
           continue
       endwhile
       if test&set(b) then
           goto loop
       endif

V(b)
       b = FALSE
       UpdateAll b to FALSE
```

Figure 2. Binary semaphore algorithm (test&set based)

---

routines arbitrate access into a region of code located between lines 6 and 7. We define a process to be within the critical section when it has finished executing line 4 in P and received a FALSE return from the test&set, but has not yet executed line 7 in V. We further assume that any process in the critical section will complete and invoke the V within a finite amount of time.

Suppose process *A* invokes P and busy-waits on the boolean semaphore b in lines 1-3. If the initial test at line 1 returns TRUE, *A* stores that value into the cache line reserved for b and continues to poll within the cache. *A* can only proceed to line 4 after a process from within the critical section invokes V and performs the UpdateAll at line 8.

Alternatively, if executing line 1 reveals a FALSE value for b, *A* must still perform the retest at 4 before proceeding into the critical section. If line 4 returns a TRUE value for b, it must be the case that some other process *B* has been granted critical section access in the interim. In this case, *A* returns to the busy-waiting loop. Note that its cached value for b has been modified to TRUE after the test&set.

We observe that the semaphore is originally FALSE and the only way it can go from FALSE to TRUE is when a process succeeds at line 4 and proceeds into the critical section. From this we derive the following invariant relation: *whenever* b*'s value* in memory *is* TRUE, *it must be the case that some process is within the critical section.*

To show that no process can be indefinitely postponed within the polling loop, we argue that any process that is polling with a TRUE value for b in its cache will eventually have its entry updated to be FALSE. Consider processes within P and the ways in which they can acquire a cache entry for b. This can occur in one of three ways: (1) as a result of executing the test&set at line 4; (2) as a result of a cache miss at line 1, or; (3) as a value left over from a previous invocation of V.

Case (1). Suppose some process *A* fetches a TRUE value for b as a result of the test&set at line 4. From the invariant relation, we know that some process *B* is simultaneously in the critical section. Thus, *A* has b in its cache and *B* has not yet executed the UpdateAll at line 8. We conclude that when *B* executes the UpdateAll it will effectively replace the stale value in *A*'s cache thereby allowing *A* to break out of the loop.

Case (2). Suppose *A* fetches a TRUE value for b from memory at line 1. At the time of that fetch, the value of b in memory must have TRUE. We can then argue as in case (1) that some process must be in the critical section and will eventually update b's representative in *A*'s cache.

Case (3). Since only FALSE values for b are installed into the cache in V, this case is not relevant.

We conclude that indefinite postponement as a result of cache polling can not occur.[5]

5.1.2. Counting semaphore

A counting semaphore is similar to a binary semaphore in that it regulates the multi-process access to a region of code. In the latter case, however, more than one process may be allowed within a *semi-critical* region. Besides being useful in their own right, these routines can also be used to construct reader-writers protocols as in [GoLR83].

[5] A more formal presentation of the arguments in this section can be found in the Appendix.

Consider the fetch&add based implementation of a counting semaphore (see Figure 3). The counter S is initialized to the maximum number of processes allowable within the semi-critical section (SCS). Only processes that are able to decrement S to a non-negative value can complete the P. Unsuccessful processes must busy-wait in lines 1-3 until another process performs the UpdateAll at line 13.

We argue against indefinite postponement in a similar manner as with the binary semaphore algorithm. In the present case, we deduce the invariant relation: *whenever* S*'s value in memory is non-positive, there must be some process in the semi-critical section.* By an analogous argument to the one in the previous section, any process looping in lines 1-3 will eventually have its cache value for S updated and can not be indefinitely postponed.

Note that in this example, the UpdateAll instruction does not try to restore a value that is necessarily consistent with main memory. It is sufficient to broadcast any positive value in order to release any processes from the busy-waiting loop. Furthermore, since this algorithm allows concurrent fetch&adds to S, it may be the case that line 9 does not even restore a positive value to the semaphore.[6] Thus, a strict adherence to consistency can lead to starvation in a pathological case.

Consider also the advantages of knowing that S is only accessed within the two routines of Figure 3. If this were not the case, we would be forced to broadcast an update each time the synchronization variable is modified (in particular, after lines 4 and 6). Since we are caching within a protected domain, we can take advantage of the fact that cache updating need only occur when a process actually leaves the semi-critical section.

```
        P(S)   { initially S = 'maximum parallelism in SCS' }
        cacheable S
1       loop:  while S <= 0
                   continue
3              endwhile
4              t = fetch&add(S, -1)
5              if t <= 0 then
6                  fetch&add(S, 1)
7                  goto loop
8              endif

        V(S)
9              fetch&add(S, 1)
10             UpdateAll S to 1
```

Figure 3. Counting semaphore algorithm (fetch&add based)

[6]For example, suppose there was one process within the SCS which allowed a maximum of two. Thus, the value of S has become 1. Two other processes proceed into the P in lock step and execute line 4 simultaneously. This results in the decrementing of S down to –1. If the process in the SCS then entered the V, it could only increment it to 0.

5.2. Barrier synchronization

Barrier synchronization is central to many parallel programming paradigms. Many forms of loop based parallelism depend upon it. The objective here is to restrain processes from proceeding past the barrier until all have reached it. Barrier algorithms are often marked by a high degree of polling as each process checks a global counter that indicates when the others have arrived. The fetch&add primitive provides an elegant means for expressing this operation [Dimi85] (see Figure 4). In our implementation, we use the single synchronization variable S, which can assume values from either of two sequences (NPES is the number of PEs assigned to the barrier):

seq_1: 0, ..., NPES
seq_2: NPES, ..., NPES*2

These sequences alternate from one to the other at consecutive instances of barrier points. The main portion of the algorithm is composed of a multiway branch (lines 2-15). The first two branches (lines 2-7) are used exclusively for seq_1 barriers, whereas the last two (lines 8-15) are used exclusively for seq_2 barriers. This partitioning of the algorithm into two phases is necessary to prevent consecutive barrier instances from interfering with one another. The two cases are symmetric and we will assume without loss of generality a seq_1 barrier. We show that processes can not be indefinitely postponed within the polling sections.

The fetch&add in line 1 keeps track of how many processes have reached the barrier. The first NPES-1 processes to reach this line increment S and assign consecutive values from the set *{* 0, ..., NPES-2 *}* to the private variable t. They then proceed to the polling loop in lines 5-7. The last process to arrive assigns the (unique) value NPES-1 to its t and proceeds to line 3. Since the fetch&add instruction imposes an ordering upon the stream

---

```
        barrier(S)   { initially S = 0 }
        cacheable S
1           t = fetch&add(S, 1)
2           if t = NPES-1 then
3               UpdateAll S to NPES
4           else if t < NPES-1
5               while S < NPES
6                   continue
7               endwhile
8           else if t = NPES*2-1
9               S = 0
10              UpdateAll S to 0
11          else    { in this case NPES-1 < t < 2*NPES-1 }
12              while S >= NPES
13                  continue
14              endwhile
15          endif
```

Figure 4. Barrier synchronization algorithm (fetch&add based)

---

of processes invoking it (the serialization principle), we derive the invariant relation: *after the last process has executed the* fetch&add*, it must be the case that all the others have already finished their respective* fetch&adds*. Therefore, they have all fetched* S *into their caches.*

Since there are no stores to S between line 1 and the the while loop (lines 5-7), the polling processes with valid cache entries for S will divide into two cases. This cache value will either be:

NPES	*if the* UpdateAll *operation has reached its cache,* or
less than NPES	*otherwise*

In the first case, the process will bypass the polling loop altogether. In the second case, the process will loop until the last process executes the UpdateAll in line 3. As soon as this operation is reflected in the cache of the polling process, the loop may be breached. Note that, as was the case for the semaphore algorithms, evictions do not pose any additional problems. If S is evicted from the cache after the UpdateAll takes effect, the PE will load it from memory where it has already been incremented to NPES by the last fetch&add (line 1). We conclude that the enhanced barrier algorithm does not allow indefinite postponement.

6. The impact of broadcasting on the network

In the previous algorithms, we have stressed the importance of limiting the number of broadcasts through software control. We are currently examining how well a multistage network responds to periodic broadcast traffic and the preliminary results support our methods.

The measurements were taken from a network-level simulator that models a buffered omega network with several user-assignable parameters including offered load, number of PEs, and frequency of broadcasts. In our experiments, offered load was fixed at 40% with the requests uniformly distributed over the possible memory range. This provided for a system with reasonably light loading in the absence of broadcasts.

As noted previously, a single broadcast operation affects a set of network switches whose number is linear in the number of PEs. It seems reasonable, therefore, to measure broadcast frequency on a system-wide basis. Thus, we factor in the number of processors when figuring the probability of a single PE making a broadcast.

The graph in Figure 5 shows how latency degrades under an increasing background of broadcast requests. The latency is shown relative to the no-broadcast state. The broadcast frequency is indicated in units that indicate the likelihood of a system broadcast relative to the offered load. Thus, a reading of k on the x-axis means that when each PE makes a request, it has a k/N probability of being a broadcast, where N is the number of processors. All other network requests were either reads or writes. The graph is centered at $k = 1$, the state in which the system has approximately one broadcast pending at every cycle.

It seems reasonable at some point to expect overall system performance to degrade in proportion to frequency of broadcasts. Note that we do not see any performance penalty until $k > .5$. At that point, the broadcast requests begin to populate buffers on the return path and latency degrades. Note that this effect is slightly more gradual in larger systems suggesting that scaling improves system performance under a fixed, system-wide frequency of broadcasts.

Clearly, the utility of broadcasts is limited to applications in which the system-wide frequency is independent of the number of PEs. This supports our methodology of tailoring the software to minimize the necessity of invoking these operations. In contrast, a general policy of using full broadcasts to ensure the cache coherency of all

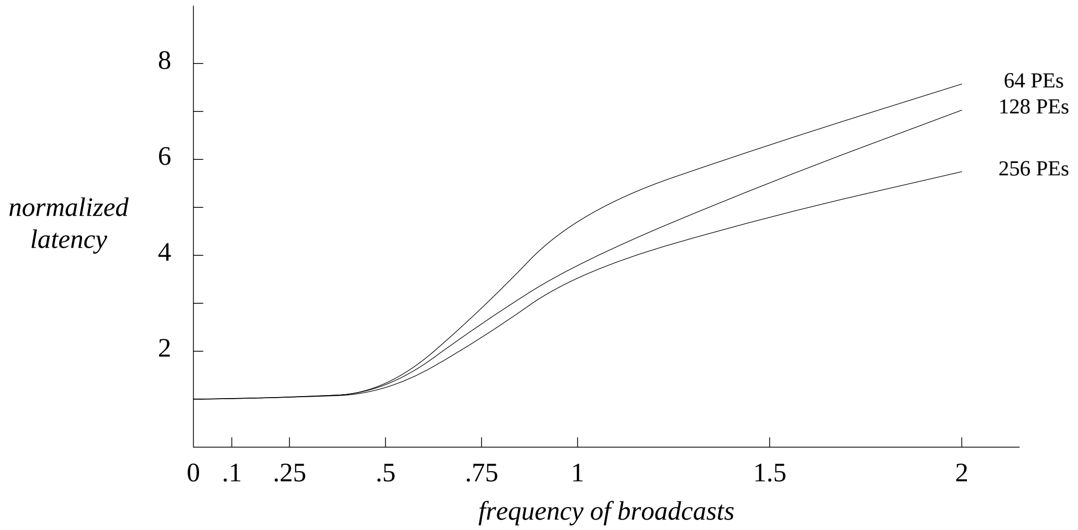

Figure 5. The effect of broadcasts on system latency

shared data is clearly infeasible in highly parallel systems. The question of using a more selective form of broadcast in this regard is an area of open research and beyond the scope of this paper.

These results also point to the possibility of pathological scenarios in using our algorithms. For example, if the number of active semaphores or barriers at any given time is proportional to the number of PEs, the broadcast traffic will saturate the system. In such cases, the system will perform better if such applications link to libraries that contain the more conventional synchronization routines and forego our technique.

7. General utility of a software accessible cache

This cache technique can be used to advantage in any program that routinely polls shared variables. An application in which this can be useful is a user-mode tasking system where a number of processes are prespawned and repeatedly invoked whenever the parent process encounters a parallel construct [Berk88]. Typically, while processes are idling they all poll a single ‘‘work available’’ flag. It is often reasonable to implement this in a busy-waiting manner, especially in a batch environment where all processors could be dedicated to each job in turn. By caching the shared flag and polling, response time can be optimized without adding to the network load.

We are also examining ways of using our technique in readers-writers lock routines. These locks provide mutual exclusion for writers and unlimited access for readers in the absence of writers. While a writer is active, we can allow readers to poll within the cache. The writer invokes UpdateAll when the write operation finishes. In a common scenario where reads are frequent and time critical and writes are less frequent but periodic, our methods serve to minimize the response time for the readers.

8. Implementation issues

As mentioned earlier, little additional hardware is needed for this kind of cache policy. There must be hardware support for propagating the updates into every processor cache, either through the network broadcast as pictured in Figure 1, or through a special purpose bus. The latter implementation has the scalability difficulties that are endemic to more generally bus based systems. Alternatively, network broadcast requires enhancing the switch nodes so that they can recognize the special UpdateAll request on the return path and propagate it to all output ports.

The modifications to the cache are more substantial but still reasonable. The control logic has to multiplex access to the tag store between the PE and the network for the implementation of broadcasting. This can be implemented either by some simple arbitration scheme or by dual-porting the cache. A duplicate tag store as in [Good83] can be employed to reduce access contention but may be unnecessary since UpdateAll operations are assumed to be infrequent. When the cache sees an UpdateAll request from the network, it modifies the cache line on a hit and does nothing on a miss.

Our original assumptions included a cache with a one word line size and physical tags. We examine the repercussions of adopting alternative design decisions. A larger line size may be accommodated if certain precautions are taken. In particular, lines that contain cacheable synchronization variables should not be written back to memory. This is to prevent potentially stale values for those variables from overwriting the correct values in memory. The simplest way of avoiding this is to isolate the cacheable synchronization variables in memory so that they occur alone within the cache line.

A virtually tagged cache presents special problems if different virtual addresses can translate to the same synchronization variable. This precludes using virtual instead of physical addresses in the UpdateAll network requests. For this reason, a simple virtual-tagged cache cannot incorporate our technique. However, a cache that augments virtual tags with physical tags [Good87] can be used in a straightforward manner. Otherwise, there would have to be some alternative means of performing fast physical-to-virtual translation (*eg*., a page frame table implemented in hardware).

It should be mentioned that it is probably not advisable to provide the UpdateAll operation for general programming use. An unrestricted ability to modify the contents of caches can lead to programming errors that are extremely hard to debug. Therefore, use of our techniques should be confined to library subroutines that can be rigorously analyzed and tested before being installed.

9. Conclusion

As multiprocessors become more prevalent, many efforts are underway to deal with the general problem of cache coherence for shared variables. Sophisticated schemes have been proposed that must be evaluated in terms of hardware cost and performance overhead. In order to keep both of these measures at reasonable levels, we have proposed a more restricted method that only extends cacheability to selected shared variables, those that are primarily used for synchronization. These variables are responsible for a disproportionately high degree of contending network traffic for many applications. Our technique of ‘‘polling the cache’’ maintains consistency by means of the UpdateAll operation, which is invoked from the software. In contrast to bus based schemes, our consistency mechanisms do not depend on the monitoring of a central resource. This makes them particularly well suited to large scale parallel machines, such as those with multistage interconnection networks. We have shown that this technique is sufficiently general to be of use in several familiar synchronization operations and in many applications that utilize polling.

References

[ASHH88]
Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz, ‘‘An Evaluation of Directory Schemes for Cache Coherence,’’ *Fifteenth Annual Symposium on Computer Architecture*, 1988, pp. 280-289.

[ArBa84]
James Archibald and Jean-Loup Baer, ‘‘An Economical Solution to the Cache Coherence Problem,’’ *Eleventh Annual Symposium on Computer Architecture*, 1984, pp. 355-362.

[Berk88]
Wayne Berke, ‘‘ParFOR — A Structured Environment for Parallel FORTRAN,’’ *Ultracomputer Note #137*, April 1988.

[CeFe78]
Lucien M. Censier and Paul Feautrier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Transactions on Computers*, C-27, December 1978, pp. 1112-1118.

[ChVe88]
Hoichi Cheong and Alexander V. Veidenbaum, ‘‘A Cache Coherence Scheme With Fast Selective Invalidation,’’ *Fifteenth Annual Symposium on Computer Architecture*, 1988, pp. 299-307.

[CyKM88]
Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe, ‘‘Automatic Management of Programmable Caches,’’ *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[DaHe88]
Helen Davis and John Hennessy, ‘‘Characterizing the Synchronization Behavior of Parallel Programs,’’ *Proceedings of ACM/SIGPLAN PPEALS*, 1988, pp. 198-211.

[Dimi85]
Isaac Dimitrovsky, ‘‘A Short Note on Barrier Synchronization,’’ *Ultracomputer System Software Note #59*, January 1985.

[GoLi73]
L. Rodney Goke and G.J. Lipovsky, ‘‘Banyan Networks for Partitioning Multiprocessor Systems,’’ *First Annual Symposium on Computer Architecture*, 1973, pp. 21-28.

[Good83]
James R. Goodman, ‘‘Using Cache Memory to Reduce Processor-Memory Traffic,’’ *Tenth Annual Symposium on Computer Architecture*, 1983, pp. 124-131.

[Good87]
James R. Goodman, ‘‘Coherency for Multiprocessor Virtual Address Caches,’’ *Proceedings of ASPLOS II*, 1987, pp. 72-81.

[Gott87]
Allan Gottlieb, ‘‘An Overview of the NYU Ultracomputer Project,’’ Ultracomputer Note #100, April 1987 (to appear in *Experimental Parallel Computing Architectures*, Jack Dongarra, ed.).

[GoLR83]
Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph, ‘‘Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors,’’ *ACM TOPLAS*, 5(2), April 1983, pp. 164-189.

[Hill86]
Mark Hill, *et. al.*, ‘‘Design Decisions in SPUR,’’ *IEEE Computer*, 19(10), November 1986, pp. 8-22.

[KrSn83]
Clyde P. Kruskal and Marc Snir, ‘‘The Performance of Multistage Interconnection Networks for Multiprocessors,’’ *IEEE Transactions on Computers*, C-32, December 1983, pp. 1091-1098.

[Lawr75]
Duncan H. Lawrie, ‘‘Access and Alignment of Data in an Array Processor,’’ *IEEE Transactions on Computers*, C-24, 1975, pp. 1145-1155.

[LiAb85]
Zhiyuan Li and Walid Abu-sufah, ‘‘A technique for reducing synchronization overhead in large scale multiprocessors,’’ *Twelfth Annual Symposium on Computer Architecture*, 1985, pp. 284-291.

[LoTh88]
Tom Lovett and Shreekant Thakkar, ‘‘The Symmetry Multiprocessor System,’’ *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[Mcau86]
Kevin P. McAuliffe, *Analysis of Cache Memories in Highly Parallel Systems*, Ph.D. dissertation, NYU, 1986.

[Pate81]
Janak H. Patel, ‘‘Performance of Processor-Memory Interconnections for Multiprocessors,’’ *IEEE Transactions on Computers*, C-30, October 1981, pp. 771-780.

[PfNo85]
Gregory F. Pfister and V. Alan Norton, ‘‘‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks,’’ *IEEE Transactions on Computers*, C-34, October 1985, pp. 943-948.

[RuSe84]
Larry Rudolph and Zary Segall, ‘‘Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,’’ *Eleventh Annual Symposium on Computer Architecture*, 1984, pp. 340-347.

[Shaw88]
Alan Shaw, *The Logical Design of Operating Systems*, Prentice Hall (second edition), 1988.

[ThGF88]
Shreekant Thakkar, Paul Gifford, and Gary Fielland, ‘‘The Balance Multiprocessor System,’’ *IEEE Micro*, (8)1, February, 1988, pp. 57-69.

[Wils87]
Andrew W. Wilson Jr., ‘‘Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors,’’ *Fourteenth Annual Symposium on Computer Architecture*, 1987, pp. 244-252.

[WuBe72]
W. A. Wulf and C. G. Bell, ‘‘C.mmp—a multi-mini-processor,’’ *Proceedings of the AFIPS*, Fall 1972, pp. 765-777.

Appendix: Proof of freedom from deadlock in binary semaphore routines

We assume that no per-processor pipelining of network requests occurs. By the serialization principle, any parallel instruction interleaving is equivalent to some serialization of the union of all instruction streams (which preserves the per-processor ordering). Hence, at each time step we may assume that exactly one PE executes exactly one machine instruction. In fact, we can assume without loss of generality that exactly one PE executes exactly one numbered statement at each time step. This derives from the fact that the proof doesn't make statements about any program variables except for the semaphore variable `b`. Since we assume that `b` is not accessed within the critical section and the statements in our algorithms each contain at most one reference to `b`, we may partially order statements from different PEs on the basis of when they access `b`. If a statement does not access `b` (eg. line 4), its interprocessor order is unconstrained. Thus, for any execution instance we can induce a total ordering upon the numbered instructions that simultaneously respects the aforementioned partial order and preserves the intraprocessor orders.

In our program model, exactly one PE executes a numbered statement at each time step. When we speak of the program state at time t, we refer to the values of all variables *before* execution of the instruction at time t. Similarly, when computing a predicate that is a function of the program state (such as the number of processes within the critical section at time t), we refer to the program state prior to the execution of the instruction at time t.

We use the following notation:

$LOC_i(t)$	is the statement number of the instruction that PE_i will next execute after time t.
$ISEXECUTING_i(t)$	is only defined when PE_i actually executes a statement at time t; when defined, it is equal to LOC_i. Note that $ISEXECUTING_i(t) = n$ implies that $LOC_i(t+1) = \mathrm{successor}(n)$ (*i.e.*, it ''bumps'' the program counter).
$MEMVAL^S(t)$	is the value of the variable S in memory at time t.
$CACHEVAL_i^S(t)$	is the value of S in the cache of PE_i at time t (if S is not cache resident at PE_i, we say that $CACHEVAL_i^S(t) = undefined$).
$INCRIT(t)$	is the number of PEs in the critical section at time t.

In the binary semaphore proof, we show that our routines are deadlock free in the context of the infinite loop presented in Figure ??. It can be easily proved that these routines ensure mutual exclusion. Further, we assume that any process within the critical section will eventually leave it.

We define PE_i to be within the critical section at time t if and only if $LOC_i(t) \in \{5, 6\}$. Thus, $0 \le INCRIT(t) \le 1$ for all t. Furthermore, $INCRIT(t) = 0$ if and only if $MEMVAL^{\mathtt{b}}(t) = FALSE$.

We demonstrate that the semaphore routines are deadlock-free by proving that, at any time during the computation, the state of the critical section ($INCRIT(t)$) must change after some finite amount of additional time.

Theorem. At any time t_0 there exists a time $\bar{t} > t_0$ such that $INCRIT(\bar{t}) \ne INCRIT(t_0)$.

```
        loop
1         L1: while b    { P(b) }
2                 continue
              endwhile
3             if test&set(b) then
4                 goto L1
              endif

5             Critical Section

6             b = FALSE    { V(b) }
7             UpdateAll b to FALSE
        endloop
```

Figure 6. Binary semaphore loop (test&set based)

Proof. If $INCRIT(t_0) = 1$, then choose $\bar{t}$ to be the time that the process within the critical section leaves it. More formally, if PE_i is in the critical section at time t_0, then $\bar{t}$ is the first $t > t_0$ *s.t.* $ISEXECUTING_i(t-1) = 6$.

If $INCRIT(t_0) = 0$, then $MEMVAL^{\mathsf{b}}(t_0) = FALSE$. The first PE to execute the test&set at line 3 after time t_0 will then enter the critical section. We can choose $\bar{t}$ to be the time immediately following the execution of this test&set. Thus, it is sufficient to show that some PE executes line 3 after t_0 as is done in the lemma below.

Lemma. If $INCRIT(t_0) = 0$, then $ISEXECUTING_i(\bar{t}-1) = 3$ for some time $\bar{t} > t_0$ and some PE_i.

Proof. We assume that no PE executes line 3 after t_0 and show how this leads to a contradiction. We first show that:

$$MEMVAL^{\mathsf{b}}(t) = FALSE \text{ for all } t \geq t_0 \qquad (\dagger)$$

This is true because $MEMVAL^{\mathsf{b}}(t) = TRUE$ implies that the value of b changed from *FALSE* to *TRUE* at some time after t_0. The only way this can occur is by some PE executing line 3, which contradicts our initial assumption.

At time t_0, each PE is either in P or will eventually enter P. In either case, in order to avoid executing line 3, each PE must eventually be within the polling loop (lines 1-2). Let τ_j be the time at which PE_j first executes line 1 after time t_0. This value is defined for all PE_j. Formally, τ_j is the first $t > t_0$ *s.t.* $ISEXECUTING_j(t) = 1$.

Consider the state of each PE_j's cache at time τ_j; in particular, the value of $CACHEVAL_j^{\mathsf{b}}(\tau_j)$. We can partition the PEs into disjoint sets based on this value.

$$\text{Let}\quad \begin{cases} \Pi_F & = \{\mathrm{PE}_j,\ s.t.\ \mathit{CACHEVAL}_j^{\mathsf{b}}(\tau_j) = \mathit{FALSE}\} \\ \Pi_T & = \{\mathrm{PE}_j,\ s.t.\ \mathit{CACHEVAL}_j^{\mathsf{b}}(\tau_j) = \mathit{TRUE}\} \\ \Pi_{undef} & = \{\mathrm{PE}_j,\ s.t.\ \mathit{CACHEVAL}_j^{\mathsf{b}}(\tau_j) = \mathit{undefined}\} \end{cases}$$

We examine each of these sets in turn under the assumption that no PE reaches line 3 and show that if any is nonempty, a contradiction occurs.

Note that any PE $\in \Pi_F$ will bypass the polling loop and go execute line 3. Furthermore, any PE $\in \Pi_{undef}$ will fetch b from memory at line 1. By (†), this value will be *FALSE* and the PE will also go on to execute line 3. We conclude that $\Pi_F \cup \Pi_{undef} = \emptyset$.

From the above, we have that $\Pi_T \neq \emptyset$ (in fact, $\mathrm{PE}_i \in \Pi_T$ for all i). Let PE_j be one such PE. The only way for PE_j to carry a *TRUE* value for b in its cache is to have fetched b in the course of executing line 3 at some time $\hat{t} < \tau_j$. This implies that $\mathit{MEMVAL}^{\mathsf{b}}(\hat{t}+1) = \mathit{TRUE}$, hence $\mathit{INCRIT}(\hat{t}+1) = 1$ and some PE_k is in the critical section at time $\hat{t}+1$. Since $\tau_j > t_0$, by (†) we must have $\mathit{INCRIT}(\tau_j) = 0$. But this implies that PE_k exited the critical section after time $\hat{t}$ and before τ_j. Consequently, $\mathit{ISEXECUTING}_k(t') = 6$ for some $\hat{t} < t' < \tau_j$. It follows that an UpdateAll b to FALSE was issued by PE_k at time t' and $\mathit{CACHEVAL}_j^{\mathsf{b}}(\tau_j) = \mathit{FALSE}$. But then $\mathrm{PE}_j \notin \Pi_T$, a contradiction. □