Appeared in ASPLOS-III, April 1989, pp. 64-75

EFFICIENT SYNCHRONIZATION PRIMITIVES FOR LARGE-SCALE CACHE-COHERENT MULTIPROCESSORS

James R. Goodman, Mary K. Vernon, and Philip J. Woest

Computer Sciences Department
University of Wisconsin - Madison
Madison, Wisconsin 53706

Abstract—**This paper proposes a set of efficient primitives for process synchronization in multiprocessors. The only assumptions made in developing the set of primitives are that hardware combining is not implemented in the interconnect, and (in one case) that the interconnect supports broadcast.**

The primitives make use of synchronization bits (syncbits) to provide a simple mechanism for mutual exclusion. The proposed implementation of the primitives includes efficient (*i.e.* local) busy-waiting for syncbits. In addition, a hardware-supported mechanism for maintaining a first-come first-serve queue of requests for a syncbit is proposed. This queueing mechanism allows for a very efficient implementation of, as well as fair access to, binary semaphores. We also propose to implement Fetch_and_Add with combining in software rather than hardware. This allows an architecture to scale to a large number of processors while avoiding the cost of hardware combining.

Scenarios for common synchronization events such as work queues and barriers are presented to demonstrate the generality and ease of use of the proposed primitives. The efficient implementation of the primitives is simpler if the multiprocessor has a hardware cache-consistency protocol. To illustrate this point, we outline how the primitives would be implemented in the Multicube multiprocessor [GoWo88].

1. Introduction

Architectural support for efficient process synchronization is an important aspect of the design of any MIMD multiprocessor. Synchronization events that occur repeatedly in parallel programs include addition and deletion of elements from a shared (work) queue, access to critical sections, enforcement of low-level data dependencies within loop iterations, and barriers. As the speed and number of component processors increase, it becomes increasingly critical to design hardware primitives that imply *minimum overhead* for these and other frequently occurring synchronization events. The goals are: (1) to minimize the number of operations required over the global interconnect for a given synchronization event, and (2) to maximize the parallelism in the execution of simultaneous synchronization requests.

In this paper we propose a set of architectural primitives, which we believe is complete for process synchronization in large-scale multiprocessors. The primitives have an efficient implementation that satisfies the above goals in multiprocessors that implement snooping or directory-based cache-coherency in hardware. We discuss how the primitives would be implemented in Multicube, a proposed shared-memory cache-coherent multiprocessor whose interconnect is a *k*-dimensional grid of broadcast buses [GoWo88, LeVe88, GoHW89].

There are three distinctive features of our proposed primitives. First, the primitives include a mechanism for first-come first-serve queueing on a semaphore. This mechanism reduces the complexity of sequentially satisfying N simultaneous requests for a semaphore, measured in number of operations over the interconnect, to $O(N)$. The best previous mechanisms for this case, based on busy-waiting using the Test&Test&Set primitive [RuSe84], require $O(N^2)$ operations over the global interconnect (see Section 2). Second, hardware Fetch_and_Φ primitives are not included in the set. The scalability of the hardware Fetch_and_Φ operations depend on hardware combining in the global interconnect, which has so far proven to be expensive. We instead propose the use of *software combining* for Fetch_and_Φ operations, and we give an example algorithm for performing the combining in software. We find that the hardware Fetch_and_Φ primitive is of little use if combining is implemented in software. Third, we propose a hardware-supported Notify primitive for global event notification. This primitive is useful for events such as barrier completion, and can be implemented efficiently if the multiprocessor's global interconnect supports broadcast, such as in Multicube.

The rest of this paper is organized as follows. Various hardware-supported primitives that have been proposed and/or implemented in particular shared-memory multiprocessors are reviewed in Section 2 to provide some background for this work. Section 3 defines the semantics of the proposed synchronization primitives. A discussion of the utility of the Fetch_and_Φ operations, and a discussion of implementing combining for these operations in software, are contained in Section 4. Section 5 presents several scenarios for using the proposed synchronization primitives in common synchronization events, demonstrating their generality and ease of use. Section 6 describes the Wisconsin Multicube, defines its architectural support for each synchronization primitive, and discusses the expected implementation costs. Many of the ideas are portable to other multiprocessor systems, with appropriate extensions to the architecture. This is discussed in Section 7. Finally, Section 8 contains a summary of this work.

2. Background

To provide some background for the synchronization primitives proposed, we review primitives that have been designed and/or implemented in particular shared memory multiprocessors.

The Sequent Symmetry multiprocessor provides three simple operations on the lowest order bit of any address in memory [Oste87]. The operations are equivalent to: Test, Test_and_Set, and Unset. The hardware required to support these primitives consists primarily of the logic for momentarily locking a cache line during the Intel 80386 "exchange-byte" (XCHB) instruction. The Symmetry primitives, together with the cache-consistency protocol, provide semi-efficient support for barrier completion testing and for mutual exclusion on critical sections. However, if N processors are spin-waiting (*i.e.* executing 'Test&Test&Set') for a lock protecting a critical section, bus traffic is $O(N^2)$ for all N processors to gain access to the lock. To see this, note that each time the lock is unset, each processor makes at least two bus accesses (one for Test and one for Test&Set), but only one processor is successful in setting the lock.

The synchronization primitives provided in the HEP multiprocessor operate on a Full/Empty bit associated with each word in memory [Jord83]. The bit is tested before a read or write operation if a special symbol is prepended to the variable name. The read or write operation blocks until the test succeeds. When the test succeeds, the bit is set to the opposite value, indivisibly with the read or write operation. These primitives are less general than read-modify-write primitives, but are more efficient for enforcing low-level *single-assignment* data dependencies across threads that have local access to a common memory. The hardware required for these primitives consists of the Full/Empty bits and the logic to initialize a bit, to queue a process if the test fails, and to implement the indivisible update operations.

The NYU Ultracomputer provides an atomic Fetch_and_Add primitive. Gottlieb, *et. al.* have shown that this can be used for synchronizing multiple readers and writers, and for managing highly parallel (work) queues [GoLR83]. This primitive is particularly interesting because the potential exists for combining simultaneous Fetch_and_Add operations on the same address into a single operation as the operations traverse the interconnect. Thus, multiple requests might be serviced in parallel. If the combining can be implemented in practice, primitives that have this property scale efficiently to large numbers of processors. The hardware required to implement Fetch_and_Add includes an adder in each memory module. In addition, hardware combining requires special, complex queueing logic at each node in the interconnection network [GGKM83].

The IBM RP3 multiprocessor and the University of Illinois Cedar multiprocessor have proposed synchronization instructions that are generalizations of the Test_and_Set and Fetch_and_Add primitives. RP3 provides seven Fetch_and_Φ operations, where Φ is one of the following functions: Add, And, Or, Min, Max, Store, or Store_if_Zero [BrMW85]. Note that it is straightforward to implement Test_and_Set using the Fetch_and_Store operation. As in the case of the NYU Fetch_and_Add primitive, the RP3 primitives require logic in the shared memory to implement the seven atomic read-modify-write operations.

Cedar provides a general atomic instruction that operates on *synchronization variables* [ZhYe87]. A synchronization variable in Cedar consists of two words: a key, and a value. The synchronization instruction has the following form: {address; (condition); operation on key; operation on value}. An asterisk may be placed on the condition to indicate that it should be tested repeatedly until it is true. This single atomic instruction is actually a set of synchronization primitives, which can be derived by specifying the operation to be performed on the key and the value [ZhYe87]. From these operations, it is straightforward to derive equivalent primitives for Test_and_Set, an indivisible Full/Empty bit test and read/write operation, and Fetch_and_Increment. For example, {X; (X.key=1)*; decrement; fetch} implements the Full/Empty bit test for a read operation. Because of the generality of the synchronization mechanism, a special processor is needed at each memory module to implement the operations.

3. Semantics of the Synchronization Primitives

This section defines a set of proposed synchronization primitives that we believe should be implemented in hardware. The primitives are defined from the programmer's point of view. The claims made in this section about the utility and generality of the primitives will be clarified in Section 5, and the claims made about the efficiency will be clarified in Section 6.

The assumptions we have made in developing the set of primitives are that the multiprocessor has shared memory, and that hardware combining is not implemented in the interconnect (see Section 4). Furthermore, the primitive in Section 3.2 assumes broadcast is supported in the interconnect.

Section 3.1 discusses *syncbits*, the data structure on which our Test_and_Set, Unset, and Queue_on_SyncBit (QOSB) primitives operate. Section 3.2 defines these three primitives, and Section 3.3 defines a fourth primitive for efficient global event notification.

3.1. Synchronization Bits (Syncbits)

The three primitives proposed in Section 3.2 operate on special synchronization bits, called *syncbits*. The syncbit enforces mutual exclusion when a particular protocol is followed by the parallel tasks. This is useful, for example, for critical sections and pairwise data sharing.

In the Multicube implementation in Section 6, we propose to associate a syncbit with each line of shared memory. (A similar idea has been suggested by Bitar and Despain [BiDe86].) That is, syncbits are addressed by addressing a line of shared memory. The term *line* in this paper implies the aligned unit of memory over which consistency is maintained.

There are at least three important advantages of associating the syncbits with lines of memory. First, synchronization memory is allocated in proportion to data memory. Second, efficient operations on the bits can be implemented as extensions to the existing cache coherence protocol. Finally, the machine can be easily programmed so that with acquisition of a semaphore (*i.e.* a syncbit), a relevant line of data immediately becomes local to the processor.

A disadvantage of associating syncbits with lines of shared memory is that care must be taken so that two data structures that require distinct syncbits are not packed into the same line. With some restrictions on the declaration of synchronization variables and their associated locks, this might be easily guaranteed by the compiler. Thus, we believe that the advantages of associating the syncbits with lines of memory outweigh the disadvantages. Also note, however, that the primitives proposed below can be implemented efficiently (but possibly with greater hardware complexity) if syncbits are allocated and addressed in some other fashion.

3.2. Test_and_Set, Unset, and Queue_on_SyncBit (QOSB)

The *Test_and_Set* operation on a syncbit address atomically sets the syncbit and returns the previous value. If the return value is "unset", the Test_and_Set operation was successful, and the issuing processor is now defined to be at the head of a FIFO queue associated with the syncbit. An *Unset* operation unsets the designated syncbit and removes the processor at the head of the syncbit queue, if the queue exists.

The QOSB (pronounced "Cosby") operation is a non-blocking operation on a syncbit address that adds the issuing processor to the syncbit queue, if the processor is not already in the queue. Once a queue has been formed, the Test_and_Set operation fails (*i.e.* returns "set") without testing and setting the syncbit, when issued by processors not at the head of the queue.

The definitions of the Test_and_Set, Unset, and QOSB synchronization primitives are summarized in Figure 1. The purpose of the QOSB primitive is that the Test_and_Set operation is highly efficient (*i.e.* nearly always completes with no operations over the global interconnect) after a processor has joined the queue. The QOSB operation generates at most one asynchronous operation over the global interconnect to put the processor in the queue. At most one additional asynchronous operation over the interconnect is required to notify the processor that it is now at the head of the queue and the syncbit is unset.

If QOSB is implemented perfectly and reliably, the scenario for using this primitive is to issue the QOSB operation to join the queue and then to spin, performing the Test_and_Set operation, until the Test_and_Set is successful.

The **Test_and_Set** operation on a syncbit address succeeds if the syncbit is "unset" and either there is no queue or the processor is currently at the head of the queue. After a successful Test_and_Set operation the issuing processor is defined to be at the head of a queue associated with the syncbit.

The **Unset** operation unsets the designated syncbit and removes the processor at the head of the syncbit queue, if a queue exists.

The **QOSB** operation is a non-blocking operation on a syncbit address that adds the issuing processor to the syncbit queue, if the processor is not already in the queue.

Figure 1. Synchronization Primitive Semantics.

Unfortunately, the implementation of QOSB in Section 6 has some probability (estimated to be extremely small) that the queue of processors waiting for a syncbit will be destroyed. In this case, the Test_and_Set and Unset operations still work correctly as defined above. However, in order to guarantee efficient (*i.e.* local) spinning, the processor must re-issue a QOSB operation on the syncbit before each Test_and_Set operation within the spin loop. This scenario is described in Section 5. The extra QOSB operation has no effect if the processor is already in the queue and the queue is still intact. If the queue has broken down, the extra QOSB operation adds the processor to a new queue for the syncbit, with no guarantee that the processor is in the same position as in the original queue.

The important property of the QOSB, Test_and_Set, and Unset operations defined above is their efficiency for lock access. When these primitives are used as described above, the number of operations over the global interconnect for N spinning processors to access a syncbit lock sequentially is $O(N)$, assuming queue breakdown does not occur. This is contrasted with the $O(N^2)$ algorithm using the Sequent shadow lock algorithm, and the higher complexity of other previously proposed primitives.

There are two other useful properties of the QOSB primitive. First, it can be used for FCFS access to binary semaphores. The first-come first-serve scheduling is slightly imperfect due to the very small probability of queue breakdown. Second, it is non-blocking, which allows a processor to execute useful instructions that are not dependent on the syncbit while it is waiting to be added to the queue (and/or to receive the notification that it is at the head of the queue).

It should be noted that a QOSB operation obligates the processor to Unset the syncbit, some time after its Test_and_Set operation succeeds, so that processors behind it in the queue will eventually obtain the syncbit. Also, QOSB and Test_and_Set operations that are issued for a syncbit by two or more processes running on the same processor may interfere with each other. However, the same algorithm which handles rebuilding of the queue also guarantees the correct handling of this case.

3.3. Broadcast Notify

Applications exist in which a number of processes wish to determine the status of an event (*e.g.* barrier completion). In a cache-coherent, shared-memory system, global event notification can be realized with conventional reads and writes to memory. Unfortunately, for many implementations, such operations generate *hot spot contention* [PfNo85], resulting in serious interconnect bottlenecks. The *Notify* primitive implements a restricted write broadcast capability to eliminate this bottleneck.

4. Fetch_and_Φ

The Fetch_and_Φ memory operation is conspicuously absent from the set of hardware-implemented synchronization primitives proposed in Section 3. This primitive is useful in many situations (*e.g.* for obtaining the next loop iteration value). However, the real power of the Fetch_and_Φ synchronization primitive is derived from the possibility of combining simultaneous Fetch_and_Φ requests into one operation that proceeds over the global interconnect to memory. With combining, the latency of a single Fetch_and_Φ operation is proportional to the path length of the combining network, and not to the number of simultaneous Fetch_and_Add requests. The best combining networks are tree-structured, having a path length of $O(\log_k N)$, where k is the (avg) degree of branching, and N is the number of processors. Thus Fetch_and_Φ operations possess the proper scaling behavior for very large multiprocessors, as contrasted with the strictly serial behavior inherent in most other synchronization primitives.

For the hardware Fetch_and_Φ primitives, combining is naturally implemented at nodes in the interconnect that forward the request to memory. Unfortunately, these combining networks are expensive, due both to the actual implementation costs and to the performance penalty for requests that don't use the combining feature. If we assume that hardware combining is too expensive, we face the following key questions. First, can the inexpensive software combining techniques proposed by Yew, Tzeng, and Lawrie [YeTL87] be applied to the combining of Fetch_and_Φ operations? Second, a simple, serial Fetch_and_Φ operation can be implemented in hardware, even if combining is not implemented for this operation. However, this primitive can also be easily implemented in software using the syncbit primitives in Section 3. If hardware combining is not implemented, is the simple hardware Fetch_and_Φ operation beneficial enough to justify its implementation complexity? We address these questions in this section. Our answer to the first question is yes, but we haven't yet devised an algorithm that we're satisfied with. Our answer to the second question is, tentatively, no.

4.1. Software Combining for Fetch_and_Add

We have investigated algorithms for implementing Fetch_and_Φ combining in software. These algorithms use the primitives proposed in Section 3, and a simple hardware Fetch_and_Φ primitive when useful.

The problem is considerably more complex than the software combining example given by Yew et. al. In their example, each processor issues exactly one request to decrement a counter, whose value will be zero when all the requests have completed. They replace the original counter with a tree of counters, and a process is assigned to exactly one of the tree's leaf nodes. Each counter in the tree is initialized to the degree of branching at that level in the tree. A process ready to perform the counter decrement operation decrements its leaf counter. If the counter is now zero, the process progresses up the tree, recursively decrementing the node counter and continuing if the counter is zero. The process that decrements the root value to zero has completed the entire operation.

Software combining for the Fetch_and_Φ operation is significantly more complex than for the above example for several reasons. First, processes repeatedly issue requests. Second, the number of processes that will issue requests within any given time frame is unknown, and each process requires a response to each request. Where the relevant workload parameters are unpredictable, there is a trade-off between how long to wait to combine requests and how quickly to respond to a single request.

An example of an algorithm that implements Fetch_and_Φ software combining, using a binary combining tree, is given in the Appendix. We are not claiming that this algorithm is optimal, but rather that it is one of the simpler algorithms we have investigated so far, and that it illustrates the use of the software combining concept for Fetch_and_Add. Other algorithms are under investigation that provide possibly higher performance and greater generality, although at an increased level of complexity. Analysis of the performance of these algorithms is also the subject of continuing study.

4.2. Simple Hardware Fetch_and_Φ

Providing simple Fetch_and_Φ operations (*i.e.* Fetch_and_Φ without combining) in hardware may reduce both the number of operations over the interconnect, and the amount of data transferred per operation, as compared with performing this operation in software. Nevertheless, the implementation of this new class of operations is probably only justified if the operations are expected to occur reasonably frequently.

We have not completely ruled out the possibility of including the hardware Fetch_and_Φ primitive in the set of primitives we recommend. However, it is currently not clear that the benefits of the primitive outweigh its implementation cost when hardware combining is too costly.

5. Scenarios

The choice of an appropriate set of synchronization primitives has been driven, so far, by the need to provide certain basic capabilities to the programmer. An alternative approach is to first choose a set of important synchronization problems, and then to find primitives that solve them. Such an approach can be used to evaluate the efficiency and ease of use of the proposed primitives, while emphasizing those solutions that will be used most extensively.

Historically, scenarios representative of a large class of synchronization problems, such as the readers-writers or the dining philosophers problem, have been used to judge synchronization primitives. However, with the provision of an efficient implementation of binary semaphores solutions to most of these problems are straightforward. Thus the efficient synchronization of large numbers of processes becomes the relevant issue. An appropriate set of additional scenarios might include simple pairwise data sharing (*e.g.* nearest neighbor communication), barrier synchronization, waiting for a global event, and work queues.

In this section we present solutions to the above problems that are applicable to shared-memory, cache-coherent multiprocessors like Multicube. These examples are written as system library routines which employ the synchronization primitives presented in Section 3. Each algorithm is evaluated in terms of the bus traffic generated and latency.

5.1. Semaphores

Syncbits and Test_and_Set are sufficient for providing a mechanism to guarantee mutually exclusive access to shared data. Executing a QOSB operation first will queue the processor for the syncbit, eliminating spinning over the global interconnect by a Test_and_Set spin loop. In addition the queue prevents starvation of processes. Since the queue mechanism provided by QOSB can be broken, it is useful to place a QOSB operation in the spin loop itself. The redundant QOSB operations are ignored, except when the queue breaks down, in which case the queue will automatically be re-built. The resulting algorithm is shown below. A simple Unset operation is used to release the lock.

```
procedure lock (addr)
begin
     QOSB (addr)
     while (TEST_AND_SET (addr)) do
          QOSB (addr)
end

procedure unlock (addr)
begin
     UNSET (addr)
end
```

The above mechanisms provide for a powerful, efficient implementation of binary semaphores. Bus traffic consists of a single QOSB operation and line transfer for each request to access a critical section. If needed data associated with the lock is placed in the same line, then the overhead of locking a line is essentially eliminated. Since the solution employs busy waiting the operating system need never be invoked. A blocking version is straightforward to implement by invoking the operating system to block the process if Test_and_Set fails after some number of iterations. The operating system could then periodically check the lock and wake up the process when it becomes available.

As with most implementations of semaphores, locks provided by Test_and_Set and QOSB are only advisory. That is, processes may read or write data protected by a lock with impunity. Only if every process follows the locking protocol can mutually exclusive access be guaranteed.

While the queueing mechanism attempts to provide first-come first-serve service, two situations make it impossible to guarantee such an ordering. First, the queue may break down, resulting in a (possibly) different order when it is rebuilt. Second, each processor is allowed only a single queue entry. When a lock arrives the first process attempting to set the lock succeeds. Any other processes on that processor will have to wait until the lock is released before another queue entry can be created.

5.2. Pairwise Data Sharing

For many applications it is important to handle efficiently a special case of mutual exclusion, namely, pairwise sharing. Since an arbitrary computation can be placed between the initial QOSB and the Test_and_Set spin loop, and since QOSB does not cause the processor to block, these primitives can be used to perform efficient prefetching.

For example, a process may QOSB for lines for each of its “nearest neighbors”, and later check if the lines have arrived, or wait on them if they have not. Thus QOSB can be used to overlap the acquisition time for multiple semaphores, implying that latency can be reduced or eliminated. This technique is useful even for lines that are not shared.

```
     . . .
QOSB (data[i+k][j])
QOSB (data[i-k][j])
QOSB (data[i][j+k])
QOSB (data[i][j-k])
     . . .
lock (data[i+k][j])
lock (data[i-k][j])
lock (data[i][j+k])
lock (data[i][j-k])
     . . .
```

Caution must be taken whenever using QOSB to perform prefetching. Issuing a QOSB request implies that the process will eventually acquire the locked line by successfully issuing a Test_and_Set, and later release it; otherwise other processes using QOSB to acquire the line will fail.

This scenario may be complicated by the possibility of a writer process that releases, re-acquires, and updates a shared line before a reader process has a chance to access the new data. This situation may occur where processes exchange data without intervening barriers. A similar situation occurs when a reader accesses the same data more than once. These cases can be solved by placing a tag in the line of the corresponding lock that each process sets before releasing the line. Now a process can spin locally until the tag has changed by repeatedly waiting for the lock, checking the tag, and releasing the lock (so another process can acquire it). This guarantees alternating access to the line. In many cases this will be unnecessary, since the queueing mechanism guarantees that a waiting processor will acquire exclusive access to a line if another processor releases it even momentarily.

```
procedure wait_turn (lock_tag)
begin
     lock (lock_tag)
     while (lock_tag == MY_PROCESS_ID) do
          unlock (lock_tag)
          lock (lock_tag)
     end
     lock_tag = MY_PROCESS_ID
end
```

5.3. Barrier Synchronization

Barrier synchronization is a mechanism which guarantees that all processes have reached a specified point in their execution before any are allowed to proceed. It is used by a large number of algorithms to synchronize loop iterations or other phases of program execution. A number of techniques for implementing barriers have been proposed including the use of special hardware [Lund87], a series of locks [Broo86], and a software combining tree with Fetch_and_Add [YeTL87].

A barrier consists of two separate functions: (1) counting the number of processes that have arrived at the barrier and (2) notifying all processes once that point has been reached. An appropriate solution to the first part is to use a software combining tree scheme, such as that described by Yew, *et. al.* [YeTL87]. The choice for the degree of the tree largely represents a trade-off between the latency due to serial fetch_and_add operations at a single node and the latency due to the logarithmic number of fetch_and_add operations which must be performed by the last process to reach the barrier.

The second step in a barrier, notifying all processes that the barrier has been reached, is an example of what we shall call *global event notification*. In multiprocessors which do not provide hardware cache coherence, notification is performed by processes spinning on some variable that is written when the barrier is reached. Of course, these accesses may be distributed by using the software combining tree to pass back the notification. However, the spinning will adversely affect the accesses over the global interconnect of those processes that have not finished.

Several solutions which require no spinning are possible. First, a simple flag can be written by the last process to reach the barrier. Since hardware cache coherency allows multiple shared copies, all spinning is performed locally. This solution is shown below.

A local temporary variable is used to hold the value of the barrier's event notification flag. Each process executes code to indicate that it has reached the barrier using a software combining tree to update the barrier count (which is a actually a tree of counters). The last process increments the event flag, while all other processes spin waiting for the event flag to change.

```
/*  Variable temp is private to each process.
Function combining_tree performs the software
combining function and returns  true  to  the
process which decrements the barrier count to
zero, and false to all others.             */

     procedure barrier (count, flag)
     begin
          temp = flag
          if (combining_tree (count)) then
               flag = temp + 1
          else
               while (flag == temp) do
                    /* spin */
     end
```

Unfortunately, the write to the event flag causes all shared copies to be invalidated, immediately after which every process will re-read the flag. If an efficient hardware mechanism exists to combine these requests [GoHW89] then this solution may be practical. However, it is also straightforward to propagate the barrier notification back through the tree by setting each node to zero and using QOSB to avoid spinning over the global interconnect.

Considering this last solution, let us assume that there are N processors involved in the barrier and that the degree of the software combining tree is D. Then the total bus traffic is dominated by the $O\left(\frac{D(N-1)}{(D-1)}\right)$ combining operations, and the latency by $\log_D N$, the height of the tree, which has $O(D \log_D N)$ serial operations in the worst case, that is, where all processes reach the barrier simultaneously.

However, in the case that all processes are waiting on a single process to finish, only $\log_D N$ serial operations are required to determine that the barrier has been reached. Thus, event notification will dominate the latency of the barrier from the point where all processes have finished performing their computations. Reducing this latency can be accomplished by providing a primitive which directly implements global event notification. This scheme substitutes the normal write to the event flag with a Notify operation which updates all shared copies, instead of invalidating them. Thus the Notify primitive avoids the read sharing problem caused by processes spinning on the event flag when it is invalidated

```
procedure barrier (count, flag)
begin
     temp = flag
     if (combining_tree (count)) then
          NOTIFY (flag, temp+1)
     else
          while (flag == temp) do
               /* spin */
end
```

Barrier synchronization is a special case of waiting for a global event. There are two other global events of particular interest:

(1) Waiting for an event that may be caused by any single process, for example, in a parallel search. The solution is particularly simple, since the processor determining that the event has occurred simply updates the event flag. A solution using Notify is shown below. It is straightforward to add a third routine which would allow processes that wish to perform computations to occasionally check for such an event.

```
procedure wait_event (flag, local_flag)
begin
     while (flag == local_flag) do
          /* spin */
     local_flag = local_flag + 1
end

procedure signal_event (flag, local_flag)
begin
     if (flag == local_flag) then
          NOTIFY (flag, local_flag+1)
     local_flag = local_flag + 1
end
```

(2) Waiting for K out of N processes to finish. The determination that the event has occurred is somewhat more complicated than in the case of barrier synchronization because all processes are not participating in the combining. Thus a simple tree algorithm is not sufficient to combine requests. Combining can be handled, however, by more general techniques, such as the software combining Fetch_and_Add algorithm in the appendix.

5.4. Work Queues

Work queues serve as a means for a collection of processes or threads to schedule work for themselves, without the overhead usually incurred when the operating system provides this function. If the unit of work is relatively small, the work queue may become a bottleneck unless multiple insertions and deletions are allowed to proceed concurrently. This is true even if the queue is the operating system ready queue.

An implementation of a work queue that eliminates serial bottlenecks has been published previously using Fetch_and_Add [GoLR83]. Unfortunately, the solution assumes hardware combining for Fetch_and_Add and results in spinning over the interconnection network by processes waiting on a full queue, an empty queue, or a queue entry that is not yet available. However, a solution requiring only local spinning is possible using the QOSB primitive and Fetch_and_Add, as demonstrated by the following scheme.

A work queue can be implemented as a circular array where each entry in the queue consists of three fields: (1) a lock for controlling insertions to that entry, (2) a lock for controlling deletions, and (3) the queue entry itself. Each of these fields must be allocated in a separate line so that actions performed on locks and queue entries do not conflict. In addition, two counters are maintained with the queue for specifying the indexes for the next insertion and next deletion.

An insert operation is performed by obtaining a unique index for insertion (modulo the queue size), using a Fetch_and_Add operation (such as that in the appendix) to increment the appropriate counter. The process then waits on the insert lock for that entry. If the last delete operation has already completed then the lock will be available and the process performing the insert will be able to proceed immediately. If not, the insert lock will be unset when the next delete operation for that entry is performed. In either case, when the lock is acquired the process is free to insert the new item into the queue, after which it will unset the delete lock for that entry. The delete operation functions similarly.

```
procedure insert (q, item)
begin
      index = fetch_and_add(q.insert,1) mod q.size
      lock (q.insert_lock[index])
      q.entry[index] = item
      unlock (q.delete_lock[index])
end

procedure delete (q, item)
begin
      index = fetch_and_add(q.delete,1) mod q.size
      lock (q.delete_lock[index])
      item = q.entry[index]
      unlock (q.insert_lock[index])
end
```

The solution requires no bounds checks, since multiple processes can be queued to perform the same operation on the same queue entry if the number of outstanding requests happens to exceed the queue length. Note that it is easy to extend the solution to allow a process to perform multiple insertions or deletions.

Each insert or delete operation requires a Fetch_and_Add, the QOSB operation requesting the appropriate lock, two line transfers (one for the lock and one for the queue entry), and an Unset operation. If software combining is employed, both bus traffic and latency will most likely be dominated by that for the fetch_and_add operation, excluding any time required waiting for a queue entry to become available.

6. Implementation of the Synchronization Primitives

The proposed synchronization primitives have been designed for implementation on a large-scale cache-coherent multiprocessor. The recently proposed Multicube architecture [GoWo88, LeVe88, GoHW89] is used as an example of such a system in order to demonstrate their implementation efficiency. This architecture is briefly described below. It should be noted that the lack of hardware cache coherency mechanisms in other multiprocessors does not preclude the use of some of the proposed primitives. This topic is left for discussion in Section 7.

The synchronization primitives take advantage of several mechanisms provided by the cache coherency hardware: (1) the ability to acquire an exclusive copy of a line, (2) the ability to locate a particular copy of a line (*e.g.* the exclusive copy), and (3) the ability to broadcast a request to all shared copies of a line. While additional hardware is required in addition to that for maintaining coherency, the provision of these three mechanisms removes the major costs associated with the implementation of the primitives.

A further characteristic of cache management is that, on a miss operation, memory space is allocated for bringing in the new line. While waiting for the line to be received the memory is unused. In addition, the copy of a line in main memory is often stale, and must not be referenced. Serendipitously, both cache and main memory contain inconsistent lines which can be so exploited almost exactly during the time that the syncbit request is enqueued. This suggests the possibility that the memory contained in inconsistent copies of a line could be used for building a queue of requesters waiting for the line.

6.1. The Multicube Architecture

The Multicube architecture employs a multi-dimensional grid of buses to provide efficient hardware cache coherency and high interprocessor bandwidth. The architecture provides for a multi-level cache structure: a first-level, or *processor*, cache for reducing memory latency and a second-level, or *snooping*, cache for minimizing bus traffic. The second level caches are envisioned as being very large (a minimum of 64 DRAMs), suggesting that for typical applications, most cache misses will result from accesses to shared data recently modified by another processor. Coherency is maintained between the two levels of cache by using a write-through strategy and imposing the MultiLevel Inclusion property [BaWa87]. Both memory and I/O devices are distributed among the processors. Because of the symmetry of the organization, bus traffic can be distributed uniformly across the buses, avoiding bottlenecks in the global interconnect. The Multicube project includes the design and implementation of a two-dimensional first generation prototype, the *Wisconsin Multicube*, shown in Figure 2.

Multicube is an attractive architecture for developing parallel applications. While providing a view of a single shared memory to the programmer, it imposes no notion of geographical locality. This ensures that applications developed for *multis* [Bell85] can be easily converted to this architecture. Thus, the Multicube is intended to be a general purpose multiprocessor architecture which supports a large range of applications, such as high-transaction database systems, large-scale simulation models, and artificial intelligence applications, as well as numerical applications.

High speed processors generally require caches to achieve high performance. In a multiprocessor, this introduces the problem of *cache coherency*. Hardware cache coherency schemes relieve the programmer and/or compiler from having to detect potential conflicts in accessing shared variables, while incurring the overhead of maintaining coherency (*i.e.* flushing cache entries to main memory) only when actually called for.

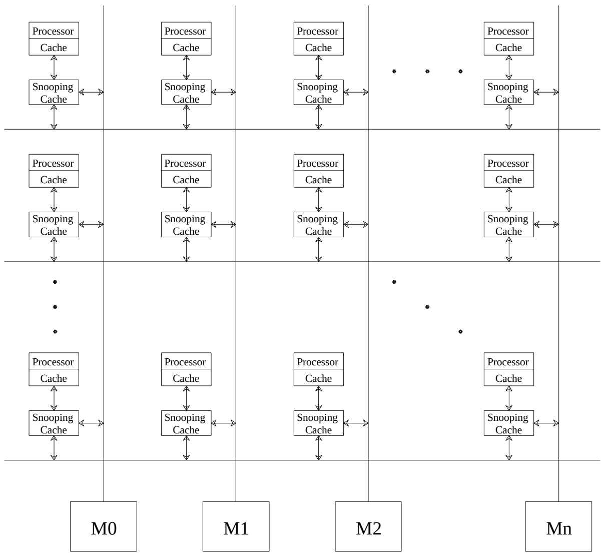

Figure 2. The Wisconsin Multicube.

The Multicube cache coherency scheme insures strict sequentiality of writes to a line by providing an exclusive copy of the line to a requesting processor. A write request that misses in the local caches results in a bus request that is either routed to the cache containing an exclusive copy of the desired line, or to main memory if the line is shared. This routing is performed by special hardware which is maintained in some type of distributed directory. If the line is currently shared when a request having the intent to modify the line reaches main memory, all outstanding copies must be invalidated. This is accomplished using a broadcast mechanism which propagates the invalidation to every processor.

6.2. Syncbits and Basic Test_and_Set

A cache line is assigned one of several states. For the basic Multicube protocol, there are only three such states: *Shared, Modified,* and *Invalid.* Globally, a memory line is always in one of two states: *Modified* or *Unmodified.* A memory line in global state Modified resides in exactly one cache, in state Modified, and is invalid in all others and main memory. A memory line in global state Unmodified is valid in main memory, and may exist in one or more caches in state Shared.

Main memory includes a tag indicating the global state of the line. A proposed technique for implementing the syncbit is to introduce additional cache states and global states, and define the syncbit in terms of a partitioning of the states. Since main memory in Multicube already maintains validity bits and possibly directory information for lines, adding states to encode the syncbit does not significantly increase the hardware complexity. Test_and_Set has the effect, then, of testing and possibly modifying the state of the cache line, including remote cache and main memory states as necessary.

The first cache state to be added is *Locked.* This state is similar to Modified in that it is held exclusively, *i.e.* it is the only copy in the system and may be written at will without generating bus traffic. It differs from Modified primarily in that the syncbit is set. Like Modified, the Locked state is both a cache state and a global state. Main memory, however, does not distinguish between the states Locked and Modified, since in neither case does it contain a valid copy of the line, and may not be informed when a change of state occurs.

In the absence of a queue, Test_and_Set atomically reads the value of the syncbit for a specified line and sets it. If the line is present locally, its state is set to Locked. A local line in Shared state must first be changed to Modified state, following the Multicube protocol. If the line is not present locally, the request is forwarded to the appropriate place: to the cache containing the Modified or Locked line, if any, and to main memory otherwise. The test is performed remotely, and if the syncbit is set, a negative response is returned. If the syncbit is unset, the protocol for changing a line to Modified is followed, and the line is returned and placed in the local cache in state Locked.

Unset, like Test_and_Set, is treated similarly to a write operation. However, in the case that the line is not present locally, the syncbit is cleared remotely. In either case, the state is changed to reflect the fact that the syncbit is unset.

6.3. Queue_On_SyncBit (QOSB)

QOSB performs two important operations: (1) it allocates space for a *shadow copy* of the line in the local cache with the *shadow syncbit* set and (2) it performs a remote access to acquire an exclusive copy of the line. Neither operation is performed if the line or a shadow is already present in the local cache. This guarantees that if a second QOSB operation is performed on the same line while the shadow line is still present, it will have no effect. It should be clear that this restriction also limits each processor to one queued QOSB request per line.

QOSB necessitates the addition of at least two additional cache states. First, a new line, in state *Shadow* is needed for indicating locally that a QOSB has occurred and a remote request for the line has been generated. In this state the data in the line is invalid, but the syncbit is set, so that a succeeding Test_and_Set operation will fail. Second, the successful completion of a QOSB requires a *Sticky* state, containing valid data, with the syncbit unset. The distinction between Sticky and Modified will be described below.

When a processor holding a line with the syncbit set receives a remote QOSB request, the request must be queued. Link information for defining the queue can be stored in shadow copies of the cache line, since the data in such lines is invalid. As the queue is built up, each shadow line is used to store a pointer to the next element of the queue. Of course the processor at the head of the queue has no such space since its copy contains valid data. Thus the queue head and tail pointers are stored in main memory where, because the line is in state Modified, the data is invalid. After the queue is created, main memory acts as the destination for succeeding QOSB requests rather than the processor containing the Locked cache line and is responsible for generating the bus operations to build the queue.

6.4. Interaction of the Primitives

Because a QOSB operation creates a shadow line in the local cache, a succeeding Test_and_Set can identify most of the circumstances under which it will fail without initiating a bus operation by simply testing the state of the local line: If the syncbit is set the Test_and_Set operation fails. This includes the cache state Shadow. If the line is present and the syncbit is unset, the Test_and_Set operation succeeds and the cache line is changed to state Locked. This includes the cache state Sticky.

Unset has an additional effect if the line is queued: It removes the head of the queue. Thus in addition to finding the Locked line, the Unset operation must initiate the transfer of the line to the next element in the queue—if one exists—where the line is placed in state Sticky. In the typical case, where the Unset acts upon a local cache line in state Locked. Note that if a queue exists, the line must be sent to main memory to be routed to the top of the queue. This inefficiency can be mitigated with a small cache that saves recent remote QOSB requests, allowing the line to be transferred directly to the appropriate cache. Of course main memory must still be notified so that it can update the head pointer.

It now becomes clear why the distinction between Modified and Sticky is necessary. If a remote QOSB request arrives for a line in Sticky state, it is queued rather than transferred as in the case of a Modified line. This is necessary to assure that no other processor is able to jump to the head of the queue and capture the syncbit between the time that the line arrives as a result of a QOSB instruction and the ensuing Test_and_Set instruction.

6.5. Implications of the Multicube Implementation

The proposed implementation contains implications for both the Test_and_Set and Unset operations. First, efficient local spinning is provided by allowing Test_and_Set to test the local shadow syncbit. Second, the hardware and software must handle the case where a shadow line is replaced in one of the caches. For hardware, this requires a broadcast mechanism to break down the queue if a queue pointer is lost. For software this requires an algorithm to rebuild the queue in the event that it breaks down. Finally, the hardware must correctly handle the case where a Locked line is replaced. Handling of this unusual case is complicated by the fact that main memory uses the buffer space for the memory line to store queue information. When a Locked line must be purged from the cache, it cannot be written to main memory without breaking down the queue. An alternative is that it could be forwarded to the cache next in the queue, where it is inserted in Locked, rather than Sticky state.

The QOSB operation imposes no responsibility on the hardware. It is simply a hint that a processor is about to perform a Test_and_Set operation. As such, it can always be ignored if necessary. Obviously serious performance degradation will result if the hint is ignored frequently, but it greatly simplifies implementation to be able to ignore it at inconvenient times.

6.6. Notify

Notify remotely writes a small number of designated bits in a cache line. In the expected case, where there are a number of shared copies distributed among the processors, the update must be propagated using a broadcast mechanism. In Multicube this can be implemented as a minor extension to the broadcast invalidate mechanism, and thus requires little additional overhead. The cache lines in state Shared to which the Notify broadcast applies are updated rather than invalidated, and remain in state Shared.

Some snooping cache protocols employ broadcast writes instead of, or in addition to, broadcast invalidations. Such protocols are known to perform efficiently for the case of a single writer and multiple readers [ArBa86], an example of which is Global Event Notification. A system that broadcasts invalidates instead of writes may benefit from this special case, though a capability for efficient read-sharing greatly reduces the benefit.

7. Porting the Primitives to Other Environments

The synchronization primitives defined in this paper were motivated by the Multicube architecture, and are well-suited for efficient implementation in that context. However, the primitives themselves assume nothing specific to Multicube, and could be implemented on any shared-memory multiprocessor, even one without hardware-guaranteed cache consistency.

As pointed out in Section 3, there are several benefits from associating a syncbit with a line of memory. In a system without caches, or with caches for which there is no hardware guarantee of consistency, the Test_and_Set, QOSB, and Unset primitives might still be appropriate, reducing interconnect traffic by eliminating non-local spin-waiting. These primitives can be realized by implementing a hardware queue in some fashion, providing each processor with a capability to determine locally if it is at the head of the queue, and providing a mechanism to notify the appropriate processor when it becomes the head of the queue.

For systems implementing hardware-guaranteed cache-consistency, the syncbit can likely be implemented by extending the cache states. The Test_and_Set, QOSB, and Unset primitives can then be implemented with the attendant benefits of associating a syncbit with a cache line. In addition, for systems employing broadcast invalidation to guarantee exclusive access for writing a line, the Notify primitive can be readily implemented as an extension to the broadcast invalidation.

8. Summary

This paper has proposed a set of efficient primitives for process synchronization in a large-scale, cache-coherent, shared-memory multiprocessor. These primitives are based on the use of synchronization bits (syncbits), logically associated with each line in memory, to provide a simple mechanism for mutual exclusion. This scheme is extended to include the use of shadow syncbits to provide for efficient (*i.e.* local) busy waiting.

A queueing mechanism that allows for an extremely efficient implementation of binary semaphores is supported. In addition an efficient global event notification mechanism is provided.

Several important synchronization scenarios such as single-reader/single-writer sharing, waiting on multiple events, barrier synchronization, and work queues, were given. These examples serve to demonstrate the efficiency, generality, and ease of use of the proposed primitives.

The Fetch_and_Add primitive is a useful mechanism for handling large numbers of processors. Unfortunately, providing scalability for this primitive through the use of hardware combining is very costly. In this paper we have proposed implementing a scalable Fetch_and_Add operation using a software combining tree and given one algorithm to illustrate the approach.

The implementation of the proposed synchronization primitives is well suited for the Wisconsin Multicube, a shared-memory cache-coherent multiprocessor. The primitives demonstrate how the Multicube's mechanisms for routing requests to a valid copy of a line, acquiring an exclusive copy of a line, and broadcasting a state change for a line can be effectively utilized. Further study has indicated that the proposed primitives may be well suited for other architectures as well.

9. Acknowledgements

We would like to thank Dave James, Pen Yew, Bob Beck, and members of the Multicube Project for many helpful discussions. Mark Hill participated in many useful discussions and provided many helpful suggestions to improve both the form and content of the paper. Ross Johnson provided important insight into Fetch_and_Add algorithms implementing software combining.

This work was supported in part by the National Science Foundation, under grants DCR-8604224 and DCR-85451405.

10. References

[ArBa86] Archibald, J., and J. L. Baer, “Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model,” *ACM Transactions on Computer Systems,* November 1986, pp. 273-298.

[BaWa87] Baer, J. L., and W. H. Wang, “Architectural Choices for Multilevel Cache Hierarchies,” *Proceedings of the 1987 International Conference on Parallel Processing,* August 1987, pp. 258-261.

[Bell85] Bell, C. G., “Multis: A New Class of Multiprocessor Computers,” *Science,* April 26, 1985, pp. 462-467.

[BiDe86] Bitar, P., and A. M. Despain, “Multiprocessor Cache Synchronization Issues, Innovations, Evolution,” *Proceedings of the 13th Annual International Symposium on Computer Architecture,* June 1986, pp. 424-433.

[BrMW85] Brantley, W. C., K. P. McAuliffe, and J. Weiss, “RP3 Processor-Memory Element,” *Proceedings of the 1985 International Conference on Parallel Processing,* August 1985, pp 782-789.

[Broo86] Brooks, E. D., “The Butterfly Barrier,” *International Journal of Parallel Programming,* August 1986, pp 295-307.

[GoHW89] Goodman, J. R., M. D. Hill, and P. J. Woest, “Scalability and Its Application to Multicube,” submitted to the *16th Annual International Symposium on Computer Architecture,* May 1989.

[GoWo88] Goodman, J. R., and P. J. Woest, “The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,” *Proceedings of the 15th Annual International Symposium on Computer Architecture,* June 1988, pp. 422-431.

[GoLR83] Gottlieb, A., B. D. Lubachevsky, and L. Rudolph, “Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors,” *ACM Transactions on Programming Languages and Systems,* April 1983, pp. 164-189.

[GGKM83] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, And M. Snir, “The NYU Ultracomputer -- Designing an MIMD, Shared Memory Parallel Machine,” *IEEE Transactions on Computers,* February 1983, pp. 175-189.

[Jord83] Jordan, H. F., “Performance Measurements on HEP -- a Pipelined MIMD Computer,” *Proceedings of the 10th Annual International Symposium on Computer Architecture,* June 1983, pp. 207-212.

[LeVe88] Leutenegger, S. T., and M. K. Vernon, “A Mean-Value Performance Analysis of a New Multiprocessor Architecture,” *Proceedings of the 1988 ACM SIGMETRICS Conference,* May 1988, pp. 167-176.

[Lund87] Lundstrom, S. F., “Applications Considerations in the System Design of Highly Concurrent Multiprocessors,” *IEEE Transactions on Computers,* November 1987, pp. 1292-1309.

[Oste87] Osterhaug, A., *Guide to Parallel Programming on Sequent Computer Systems,* 2nd ed., Sequent Computer Systems, Inc., Beaverton, Oregon, 1987.

[PfNo85] Pfister, G. A., and V. A. Norton, “Hot Spot Contention and Combining in Multistage Interconnection Networks,” *Proceedings of the 1985 International Conference on Parallel Processing,* August 1985, pp. 790-797.

[RuSe84] Rudolph, L., and Z. Segall, “Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,” *Proceedings of the 11th Annual International Symposium on Computer Architecture,* June 1984, pp. 340-347.

[YeTL87] Yew, P. C., N. F. Tzeng, and D. H. Lawrie, “Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,” *IEEE Transactions on Computers,* April 1987, pp 388-395.

[ZhYe87] Zhu, C. Q., and P. C. Yew, “A Scheme to Enforce Data Dependence on Large Multiprocessor Systems,” *IEEE Transactions on Software Engineering,* June 1987, pp. 726-739.

Appendix

A Software Combining Tree Implementation of a Combining Fetch_and_Add Operation

An algorithm for providing combining Fetch_and_Add in software is described below. In this algorithm, requests to increment a shared counter are made at arbitrary times by a set of processes. The *counter* variable is structured as a binary software combining tree with separate increments stored at each node used for combining, and the actual value stored at the root.

The Fetch_and_Add operation consists of three distinct phases which correspond to a process (1) moving up the tree "claiming responsibility for" individual nodes, (2) revisiting the claimed nodes to perform combining, and (3) waiting for and then distributing the results to those nodes where combining has been performed.

Each node in the tree consists of five fields: *status*, *wait_flag*, *first_incr*, *second_incr*, and *result*. The field *wait_flag* indicates if a process is waiting for a result at that node. *first_incr* is the amount the subtree containing the process that has claimed the node intends to increment the counter by; *second_incr* is the amount the waiting process intends to increment the counter by. Finally *result* is the counter value to be distributed down the next (sub)tree.

The *status* field designates what state the node is in. The root node is always in state ROOT. Other nodes can have one of three possible values, corresponding to the which phase of the algorithm the node is participating in:

(1) FREE: this node is unclaimed;

(2) COMBINE: this node is for combining;

(3) RESULT: this node contains results.

The algorithm proceeds as follows. (See Figure A.2.) In Part One, A process progresses up the combining tree marking each FREE node as a COMBINE node. If the process finds a RESULT node it must wait until the previous Fetch_and_Add operation finishes using this node (*i.e.* the node will become either FREE or COMBINE), before continuing up the tree. When a ROOT or COMBINE node is found, this node is locked, and the algorithm continues to Part Two.

In Part Two the process locks each node previously visited, bottom-up, and tallies the node *second_incr* values, which may have been updated since the node was first visited. Along the way, the tally for the previous subtree is stored in *first_incr*. The total tally represents the aggregate increment requested by the subtree the process is responsible for. The revisited nodes will remain locked until results are distributed.

In Part three, if a COMBINE node was reached then the final tally is added to *second_incr* for that node, the *wait_flag* field for the node is set to true, and the process spins on the status field (using the pairwise-sharing algorithm of Section 5) until the node becomes a RESULT node. For either a RESULT or ROOT node the *result* of the node is saved for distributing results downward. In the case of the ROOT node the result value must be incremented by the total tally, essentially performing the Fetch_and_Add value on the "actual counter".

The algorithm then enters Part Four, where the process reverses it's path down the tree, distributing results. At each node, if there is a waiting process, the node's *result* field is set to the *result* from Part Three plus it's own subtree's increment (*i.e. first_incr*), and the node *status* is set to RESULT. Otherwise, the node is re-initialized to FREE.

Figure A.1 shows an example of increment requests (on the arcs) and initial result values (in the nodes) that propagate down the tree for one request that reaches the root of a binary software combining tree. The process that claims each node is indicated by the bold-face path of arcs below the node. It's combined increment request is the value on the bold incoming arc. Note that the initial result value in each node has not been incremented by either of the subtree requests for the sake of clarity.

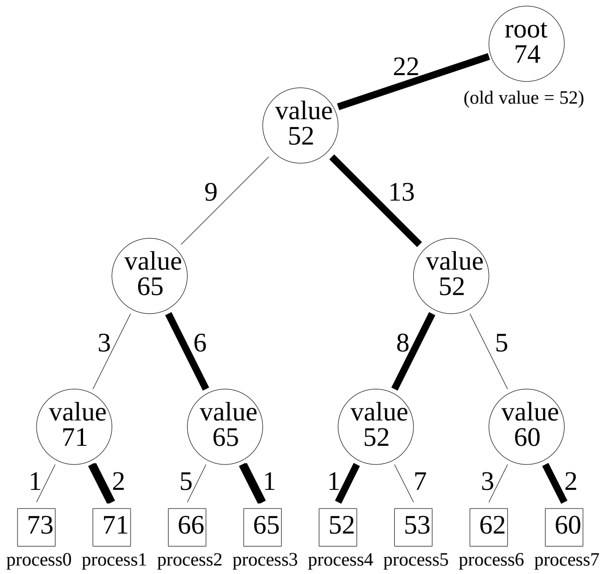

Figure A.1. Example Binary Software Combining Tree. Request from processor 4 reaches the root node. Arcs are labeled with combined increment request values. Initial return values are given in the nodes. The first subtree to be given a result is indicated by a bold arc.

```
function fetch_and_add (counter, incr)
begin
```

/* Part One. Go up the tree changing FREE nodes to COMBINE nodes (and releasing them), until a ROOT node or COMBINE node is found. If a RESULT node is encountered, spin wait until its status changes before continuing up the tree. Function node_addr returns the address of a node in the tree (counter) for a given process and level. The level above the processors is the lowest numbered level. */

```
level = FIRST_LEVEL
going_up = TRUE
while (going_up)
    node = node_addr(counter,level,pid)
    lock (node)
    if (node.status == RESULT) then
        unlock (node)
    else if (node.status == FREE) then
        node.status = COMBINE
        unlock (node)
        level = level + 1
    else /* COMBINE or ROOT node */
        last_level = level
        going_up = FALSE
end
```

/* Part Two. Go back through the nodes, first prefetching them. Then lock each node and perform the combining at each level. The nodes remain locked until results are to be distributed. Note that the value assigned to first_incr is the total from the previous level. */

```
for level = FIRST_LEVEL to last_level-1 do
    visited = node_addr(counter,level,pid)
    QOSB (visited)
end
total = incr
for level = FIRST_LEVEL to last_level-1 do
    visited = node_addr(counter,level,pid)
    lock (visited)
    visited.first_incr = total
    if (visited.wait_flag) then
        total = total + visited.second_incr
end
```

/* Part Three. If Part One stopped at a COMBINE node then place the total for this process into second_incr for the node, set the wait_flag, and wait for the node status to change to RESULT. When results are available the result is saved and the node is set to FREE. If Part One stopped at the ROOT node then saves the result and add the total in. This step performs the fetch_and_add on the actual counter value. For both cases the node should then be released. */

```
 if (node.status == COMBINE) then
     node.second_incr = total
     node.wait_flag = TRUE
     while (node.status == COMBINE) do
         unlock (node)
         lock (node)
     end
     node.wait_flag = FALSE
     node.status = FREE
     saved_result = node.result
 else /* ROOT node */
     saved_result = node.result
     node.result = node.result + total
 end
 unlock (node)
```

/* Part Four. Walk back down the tree, either freeing nodes or distributing results if combining was performed at this node (*i.e.* wait_flag is set). The result left in each node is the save result from Part Three incremented by first_incr, which is the total from the subtree for which this process is responsible. Finally, the saved result from Part Three is returned by the fetch_and_add algorithm. */

```
 for level = last_level-1 to FIRST_LEVEL do
     visited = node_addr(counter,level,pid)
     if (visited.wait_flag) then
         visited.status = RESULT
         visited.result = saved_result +
                          visited.first_incr
     else
         visited.status = FREE
 end
 return (saved_result)
end /* fetch_and_add */
```

Figure A.2. The Software Combining Fetch_and_Add Algorithm.