Fast, Contention-Free Combining Tree Barriers

Michael L. Scott
John M. Mellor-Crummey

CRPC-TR92221
June, 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Fast, Contention-Free Combining Tree Barriers

Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY 14627–0226
scott@cs.rochester.edu

John M. Mellor-Crummey
Computer Science Department
Rice University, P.O. Box 1892
Houston, TX 77251
johnmc@cs.rice.edu

June 1992

Abstract

Counter-based algorithms for busy-wait barrier synchronization execute in time linear in the number of synchronizing processes. This time can be made logarithmic in the number of processes by adopting algorithms based on trees or FFT-like synchronization patterns. As an additional improvement, Gupta and Hill [5] have proposed an *adaptive combining tree* barrier that exploits non-uniformity in inter-barrier computation times: processes begin to leave the barrier in time logarithmic in the number of processes when all processes arrive at once, but in constant time after the arrival of the last process when arrival times are skewed. Building on earlier work [4], Gupta and Hill present both regular and *fuzzy* versions of their barrier. The fuzzy version allows a process to perform useful work between the point at which it notifies other processes of its arrival at the barrier and the point at which it waits for all other processes to arrive.

Unfortunately, like many forms of busy-wait synchronization, adaptive combining tree barriers as originally devised can induce large amounts of memory and interconnect contention in shared-memory multiprocessors, seriously degrading performance. They also perform a comparatively large amount of work at every tree node, raising the possibility that the constant factors in their execution time may be unacceptably high on machines of reasonable size. To address these problems, we present a new adaptive combining tree barrier, with fuzzy variant, that achieves significant speed improvements by spinning only on locally-accessible locations, and by using atomic `fetch_and_store` operations to avoid explicit locking of tree nodes. We also present a version of this barrier (again with fuzzy variant) that employs breadth-first wakeup of processes to reduce context switching when processors are multiprogrammed. We compare the performance of these new algorithms to that of other fast barriers on a 64-node BBN Butterfly 1 multiprocessor and on a 35-node BBN TC2000. Results suggest that adaptation is of little benefit, but that the combination of fuzziness with tree-style synchronization is of significant practical importance: fuzzy combining tree barriers with local-only spinning outperform all known alternatives on the TC2000 when the amount of fuzzy computation exceeds about 10% of the time between barriers.

This work was supported in part by the National Science Foundation under Institutional Infrastructure grant CDA-8822724 at the University of Rochester and Cooperative Agreements CCR-8809615 and CCR-9120008 at Rice University. Some of the experiments described in this paper were performed on a BBN TC2000 that is part of the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory.

1. Introduction

A *barrier* is a synchronization mechanism that ensures that no process advances beyond a particular point in a computation until all processes have arrived at that point. Barriers are widely used to delimit algorithmic phases; they might guarantee, for example, that all processes have finished updating the values in a shared matrix in step t before any processes use the values as input in step $t+1$. If phases are brief (as they are in many applications), barrier overhead may be a major contributor to run time; fast barrier implementations are thus of great importance. This paper focuses on busy-wait (spinning) barrier implementations for shared-memory multiprocessors. It also considers non-spinning (scheduler-based) implementations when processors are multiprogrammed.

In the simplest barrier algorithms, each process increments a shared, centralized counter as it reaches a barrier, and spins until that counter (or a flag set by the last arriving process) indicates that all processes are present. Such centralized algorithms suffer from several limitations:

linear asymptotic latency

On a machine without hardware combining of atomic instructions, achieving a barrier requires time linear in the number of processes, P. Specifically, it requires a sequence of $O(P)$ updates to the central counter followed (in the absence of broadcast) by $O(P)$ reads.

contention

Because processes spin on a central location, traditional centralized barriers can generate contention for memory and for the processor-memory interconnection network. Such contention degrades the performance of any process that initiates references involving the processor interconnect or a saturated memory bank.

unnecessary waiting

Processes that arrive at a barrier early (to announce to their peers that they have completed some critical computation) must wait for their peers to arrive as well, even if they have other work they could be doing that does not depend on the arrival of those peers.

To improve asymptotic latency, several barriers have been developed that run in time $O(\log P)$. Most use some form of tree to gather and scatter information [6, 10, 14, 16]; the butterfly and dissemination barriers of Brooks [2] and of Hensgen, Finkel, and Manber [6] use a symmetric pattern of synchronization operations that resembles an FFT or parallel prefix computation. The butterfly and dissemination barriers perform a total of $O(P \log P)$ writes to shared locations, but only $O(\log P)$ on their critical paths. The various tree-based barriers perform a total of $O(P)$ writes to shared locations, with $O(\log P)$ on their critical paths. On most machines, logarithmic barriers can be designed to eliminate contention by having processes spin only on locally-accessible locations (either in a local coherent cache, or in a local portion of shared memory) [14].

To reduce unnecessary waiting at barriers, Gupta introduced the notion of a *fuzzy barrier* [4]. A fuzzy barrier consists of two distinct phases. In the first phase, processes announce that they have completed all the work on which their peers depend. In the second phase they wait until all their peers have made similar announcements. A traditional centralized barrier can be modified trivially to implement these two phases as separate `enter_barrier` and `exit_barrier` routines. Unfortunately, none of the logarithmic barriers mentioned above has such an obvious fuzzy version.[1]

[1] In the butterfly and dissemination barriers, no process knows that the barrier has been achieved until the very end of the algorithm. In a static tree barrier [14], and in the tournament barriers of Hensgen, Finkel, and Manber [6] and Lubachevsky [10], static synchronization orderings force some processes to wait for their peers before announcing that they have reached the barrier. In all of the tree-based barriers, processes waiting near the leaves cannot discover that their peers have reached the barrier until processes higher in the tree have already noticed this fact.

Logarithmic barriers also introduce a pair of additional problems:

lack of amortization

The critical path requires $O(\log P)$ writes to shared locations *after the arrival of the last process* before any process can continue. In a traditional centralized barrier, the last arriving process discovers that the barrier has been achieved in constant time (ignoring possible delay due to contention).

deterministic ordering

If processors are multiprogrammed, Markatos et al. [11] have shown that deterministic ordering of process synchronizations can cause an unreasonably large number of context switches. (Busy-waiting remains an option in the presence of multiprogramming so long as each process will eventually relinquish the processor when the condition for which it is waiting is not satisfied.) In the worst case, a busy-waiting tree-based barrier may require $O(P)$ context switches at every level of the tree. Even in the best case, or with scheduler-based synchronization, more than P context switches will be required in every barrier episode.

The combining tree barrier of Yew, Tzeng, and Lawrie [16] has less deterministic ordering than other tree barriers; it uses atomic `fetch_and_add` instructions in each node of the tree to elect the process that will continue up to the parent. Once the arrival phase of the barrier is complete, however, the processes waiting on any particular path to the root must be awoken in LIFO order.

To address the lack of amortization in logarithmic barriers, Gupta and Hill [5] introduced the concept of an *adaptive combining tree* barrier. Each process arriving at an adaptive combining tree barrier performs a local modification to the tree that allows later arrivals to start their work closer to the root. Given sufficient skew in the arrival times of processes, the last arriving process performs only a constant amount of work before discovering that the barrier has been achieved. To address unnecessary waiting, Gupta and Hill also devised a fuzzy version of their algorithm, with a separate tree traversal for the wakeup phase of the barrier.

In comparison to other tree-based barriers, both adaptiveness and the separate wakeup traversal (for fuzziness) introduce an as-yet unquantified amount of overhead. We have studied and refined combining tree barriers in an attempt to evaluate and minimize this overhead, eliminate deterministic ordering, and measure performance with respect to other known fast barriers. Specifically, we present

(1) an adaptive combining tree barrier, with fuzzy variant, that reduces overhead by spinning only on locally-accessible locations, and by using atomic `fetch_and_store` operations[2] to avoid explicit locking of tree nodes;

(2) another adaptive combining tree barrier, with fuzzy variant, that reduces context switches on a multiprogrammed machine by enabling a process to continue execution once any running process knows that the barrier has been achieved; and

(3) a set of experiments comparing the performance of the various adaptive (and non-adaptive) combining tree barriers to that of the fastest known centralized and logarithmic barriers, on a 64-node BBN Butterfly 1 machine and a 35-node BBN TC2000.

We conclude that adaptation is of little benefit, but that the combination of fuzziness with tree-style synchronization is of significant practical importance: fuzzy combining tree barriers with local-only spinning are competitive with the best known algorithmic alternatives, and appear in

[2] `Fetch_and_store (L, V)` returns the value in location `L` and replaces it with `V`, as a single atomic operation.

fact to be the fastest barrier for modern processors when the amount of fuzzy computation exceeds about 10% of the time between barriers. In addition, the use of breadth-first wakeup in a fuzzy conbining tree barrier significantly reduces the number of context switches, and hence execution time, when processors are multiprogrammed.

We review Gupta and Hill's adaptive combining tree barrier in section 2. We present new algorithms in section 3, performance results in section 4, and conclusions in section 5.

2. Previous Algorithms

Gupta and Hill's adaptive combining tree barrier appears in figure 2. The algorithm employs two instances of the barrier data structure for use in alternating barrier episodes. An initialization routine (not shown) establishes each data structure as a binary tree of nodes, with one leaf for every process. The `reinitialize` routine (called but not shown) restores the `left, right, parent, visited,` and `notify` fields of a node to their original values.

To take part in a barrier episode, a process starts at its leaf and proceeds upward, stopping at the first node (*w*) that has not been visited by any other process.[3] It then modifies the tree (see figure 1) so that *w*'s other child (*o*, the child through which the process did not climb) is one level closer to the root. Specifically, the process changes *o*'s parent to be *p* (the parent of *w*), and makes *o* a child of *p*. A process that reaches *p* through *w*'s sibling will promote *o* another level, and a later-arriving process, climbing through *o*, will traverse fewer levels of the tree than it would have otherwise.

A process that finds that its leaf has a nil parent knows that it is the last arrival, and can commence a wave of wakeups. It sets the `notify` flag in the root of the tree. The process waiting at the root then sets the `notify` flags in the root's children, and so on. Each process on its way out of the tree reinitializes its leaf and the node at which it waited. Two instances of the barrier data structure are required to ensure that no process can get to the next barrier episode and see nodes that have not yet been reinitialized in the wake of the previous episode.

The key to the correctness of Gupta and Hill's algorithm is its synchrony: no two processes ever see changes to the tree in an inconsistent order. In the initial loop, for example, one might think that a process that finds that *w* has already been visited could simply proceed to *w*'s parent. Allowing it to do so, however, would mean that a process might discover that the barrier has been achieved while some of its peers are still adapting nodes farther down in the tree. These adaptations could then interfere with node reinitializations during wakeup. In a similar vein, the lock on *o* in the second loop ensures that *o*'s pointer to *p* and *p*'s pointer to *o* are changed mutually and

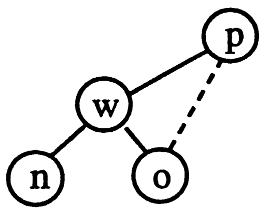

Figure 1: Naming of nodes in the adaptive combining tree barrier.

[3] A program that wishes to change the set of processes that are to take part in a given barrier episode must modify the barrier's data structures accordingly. The complexity of these modifications is a weakness shared by all of the logarithmic time barriers.

```
type node = record
    lock : syncvar := free
    visited : (no, left, right) := no
    root, bin_left, bin_right, bin_parent : ^node := // tree
    left, right, parent : ^node := bin_left, bin_right, bin_parent
    notify : Boolean := false

type instance = record
    my_leaf : ^node := // as appropriate, in tree

private instances : array [0..1] of instance
private current_instance : ^instance := &instances[0]

procedure barrier ()
    n : ^node := current_instance^.my_leaf
    loop
        w : ^node := n^.parent
        if w = nil                      // We are the last arrival.
            n^.root^.notify := true
            reinitialize (n)
            current_instance := if current_instance = &instances[1]
                then &instances[0] else &instances[1]
            return
        acquire (w^.lock)
        if w^.visited = no
            exit loop
        release (w^.lock)
    reinitialize (n)
    w^.visited := if w^.left = n then left else right
    release (w^.lock)

    // adapt tree:
    loop
        o : ^node := if w^.visited = left then w^.right else w^.left
        acquire (o^.lock)
        if o^.visited = no
            exit loop
        release (o^.lock)
    p : ^node := w^.parent
    if p <> nil
        (if p^.left = w then p^.left else p^.right) := o
    o^.parent := p
    release (o^.lock)

    // wait for barrier to be achieved:
    repeat until w^.notify              // spin

    // notify descendants:
    w^.bin_left^.notify := true         // unnecessary but harmless if
    w^.bin_right^.notify := true        // children are leaves
    reinitialize (w)
    current_instance := if current_instance = &instances[1]
        then &instances[0] else &instances[1]
```

Figure 2: Gupta and Hill's adaptive combining tree barrier.

atomically. Both loops release any node (*w* or *o*) that is found to have been visited already, in the knowledge that some other process will replace the pointer to it with a pointer to some unvisited node. Both loops therefore embody spins, and may execute an unbounded number of times.

In constructing a fuzzy version of their adaptive combining tree barrier, Gupta and Hill observed that a simple separation of the arrival and wakeup phases does not suffice to minimize unnecessary waiting: processes may not call `exit_barrier` in the same order they called enter_barrier. Processes that acquired nodes near the fringe of the tree in enter_-barrier, but which call exit_barrier early may have to wait (needlessly) for processes that acquired nodes near the root of the tree in enter_barrier, but which call exit_barrier relatively late. The solution to this problem is to employ a separate tree traversal in the wakeup phase of the algorithm, so that processes that call exit_barrier early busy-wait on nodes that are close to the root.

Code for a modified version of Gupta and Hill's fuzzy adaptive combining tree barrier appears in figure 3. We have broken out the recursive part of the wakeup phase as a separate routine to make the use of alternating trees explicit. We have also introduced changes to address some subtle bugs in the original version that are not obvious on paper, but which emerged in the course of experimentation:

(1) set *n*'s occupied flag in the first if statement of rec_exit, even if *n*'s notify flag is already set. This prevents a process from entering a node and setting the notify flags of children when some other process has already returned from that node and reinitialized one of the children.

(2) set *p*'s last_visitor field in the third if statement, prior to releasing the lock on *n* and moving (recursively) up the tree. This ensures that when a process makes a recursive call up into a parent node, no other process will reinitialize that node out from under it (the other process must first acquire the locks on the children).

(3) introduce left_notify and right_notify flags to ensure that no node is notified more than once. Without these flags it is possible for a process to climb into a node, notice that it has been notified, reinitialize it, and return, while another process higher in the tree is about to notify it again. If the second process then returns into the re-notified node, it will enter an infinite wait, expecting some other process to clear the notify flag.

(4) wait for *n* to be reinitialized by a more recently-arriving process only after last_visitor has stabilized. After the series of nested elses in rec_exit we know that the barrier has been achieved. After the if not leaf clause near the end of rec_exit we know that no other process will climb through node *n* in the future. If no other process has climbed through after us, then we can safely reinitialize *n*. Otherwise, we wait for the last process that got through to reinitialize it. It is safe to execute the last if statement after *n* has been reinitialized: n^.last_visitor will be unequal to anybody's pid, and n^.notify will be false.

In the original version of the algorithm, the wait for reinitialization appeared in a separate if n^.last_visitor <> pid clause immediately after the nested elses—before the setting of child notify flags. The wait could inadvertently be skipped if a later-arriving process had not yet set n^.last_visitor; we could therefore return into a child and reinitialize it before the late-arriving process set the child's notify flag.

3. New Algorithms

It is well known that contention for memory locations and processor-memory interconnect bandwidth seriously degrades the performance of traditional busy-wait synchronization algorithms. Previous work has shown how to eliminate this contention for mutual exclusion locks, reader-writer locks, and barriers [1,3,8,9,13-15]. The key is for every process to spin on separate *locally-accessible* flag variables, and for some other process to terminate the spin with a single remote write operation at an appropriate time. Flag variables may be locally-accessible as a result of coherent caching, or by virtue of allocation in the local portion of physically distributed shared memory. Our experience indicates that the elimination of remote spinning can yield dramatic performance improvements.

```
type node = record
    lock : syncvar := free
    visited : (no, left, right) := no
    last_visitor : pid := none
    root, bin_left, bin_right, bin_parent : ^node := // tree
    left, right, parent : ^node := bin_left, bin_right, bin_parent
    left_notify, right_notify : Boolean := false, false
        // added to prevent multiple notifies
    notify, occupied : Boolean := false, false

type instance = record
    my_leaf : ^node := // as appropriate, in tree

private instances : array [0..1] of instance
private current_instance : ^instance := &instances[0]

procedure enter_barrier ()
    n : ^node := current_instance^.my_leaf
    loop
        w : ^node := n^.parent
        if w = nil
            n^.root^.notify := true
            return
        acquire (w^.lock)
        if w^.visited = no
            exit loop
        release (w^.lock)
    w^.visited := if w^.left = n then left else right
    release (w^.lock)

    loop
        o : ^node := if w^.visited = left then w^.right else w^.left
        acquire (o^.lock)
        if o^.visited = no
            exit loop
        release (o^.lock)
    p : ^node := w^.parent
    if p <> nil
        (if p^.left = w then p^.left else p^.right) := o
    o^.parent := p
    release (o^.lock)

procedure rec_exit (n : ^node)
// n^.lock is held, and n^.last_visitor = my_pid
    if n^.notify
        n^.occupied := true              // missing in original
        release (n^.lock)
    else
        p : ^node := n^.bin_parent
        if p = nil                       // n is the root
            n^.occupied := true
            release (n^.lock)
            repeat until n^.notify       // spin
        else
            acquire (p^.lock)
            if not p^.occupied
                p^.last_visitor := my_pid
                release (n^.lock)        // before previous line in original
                rec_exit (p)             // recursive call
            else
                release (p^.lock)
                n^.occupied := true
                release (n^.lock)
                repeat until n^.notify  // spin
        // At this point we know the barrier has been achieved.
        if not leaf (n)
            // signal children ONCE AND ONLY ONCE
            acquire (n^.bin_left^.lock)
            if n^.last_visitor = my_pid and n^.left_notify = false
                n^.left_notify := true
                n^.bin_left^.notify := true
            release (n^.bin_left^.lock)
            acquire (n^.bin_right^.lock)
            if n^.last_visitor = my_pid and n^.right_notify = false
                n^.right_notify := true
                n^.bin_right^.notify := true
            release (n^.bin_right^.lock)
            // The left_notify and right_notify flags prevent multiple notifies
            // of the same child, which could otherwise lead to an infinite wait.
        // At this point if n^.last_visitor = my_pid it will stay so.
        if n^.last_visitor = my_pid
            reinitialize (n)
        else
            repeat while n^.notify
            // Wait until reinitialized; that way we don't return into a child
            // and reinitialize it before its notify flag gets set.
            // This line was originally before the sets of child notify flags.

    procedure exit_barrier ()
        n : ^node := current_instance^.my_leaf
        acquire (n^.lock)
        n^.last_visitor := my_pid
        rec_exit (n)
        current_instance := if current_instance = &instances[1]
            then &instances[0] else &instances[1]
```

Figure 3: Gupta and Hill's fuzzy adaptive combining tree barrier (modified).

We present our barriers in pseudo-code below; complete C versions for the BBN Butterfly 1 and the BBN TC2000 can be obtained via ftp (login 'anonymous', password anything) from cayuga.cs.rochester.edu (directory pub/scalable_synch/adaptive).

3.1. A Local-Spinning Adaptive Combining Tree Barrier

To eliminate remote spinning in the (non-fuzzy) adaptive combining tree barrier of figure 2, we must address three sources of spinning on remote locations:

(1) While waiting for the barrier to be achieved, processes spin on a flag in a dynamically-chosen tree node.

(2) In order to ensure consistent modifications to the tree, processes acquire and release `test_and_set` locks in every node they visit.

(3) In both the original search for a parent node at which to wait, and in the subsequent search for a sibling node whose parent should be changed, processes spin until they succeed in locking the node they are looking for *and* find it to be unvisited.

We eliminate the first type of remote spinning by using a statically-allocated per-process flag, and storing a pointer to this flag in the dynamically-chosen tree node. We eliminate the second and third types of remote spinning by using `fetch_and_store` instructions to modify the tree in an asynchronous, *no-wait* fashion [7], thereby eliminating locks. (Contention-free spin locks would eliminate the second kind of remote spin, but not the third. With the third kind of spin, the number of remote references per processor in a barrier episode has no fixed bound. The no-wait

```
shared pseudodata : Boolean

type node = record
    visitor : ^Boolean := &pseudodata
    bin_left, bin_right, bin_parent : ^node := // tree
    depth : integer := // as appropriate, in unmodified tree
    inorder : integer := // as appropriate, in unmodified tree,
        // to determine left and right descendancy
    left, right, parent : ^node := bin_left, bin_right, bin_parent

type instance = record
    f : Boolean := false
    root, my_leaf, my_internal_node : ^node := // as appropriate, in tree
        // my_internal_node is used only for reinitialization

private instances : array [0..2] of instance
    // assumed to lie in memory accessible to other processes
private current_instance : ^instance := &instances[0]
private previous_instance : ^instance := &instances[2]

procedure barrier ()
    // find place to wait:
    n : ^node := current_instance^.my_leaf
    loop
        w : ^node := n^.parent
        if w = nil
            // signal achievement of barrier
            current_instance^.root^.visitor^ := true
            goto rtn
        x : ^node := fetch_and_store (&w^.visitor, &current_instance^.f)
        if x = &pseudodata
            exit loop
        w^.visitor := x     // already visited; put it back
        n := w

    // adapt tree:
    o : ^node := if n^.inorder < w^.inorder then w^.right else w^.left
    p : ^node := w^.parent
    if p = nil
        o^.parent := nil
    else
        (if w^.inorder < p^.inorder then p^.left else p^.right) := o
        loop
            t : ^node := fetch_and_store (&o^.parent, p)
            if t^.depth > p^.depth        // swap was a good thing
                exit loop
            p := t

    // await notification and pass on the news
    repeat until current_instance^.f      // spin

    w^.bin_left^.visitor^ := true
    w^.bin_right^.visitor^ := true

 rtn:
    reinitialize (previous_instance)
    previous_instance := current_instance
    current_instance := if current_instance = &instances[2]
        then &instances[0] else current_instance + 1
```

Figure 4: An adaptive combining tree barrier with local-only spinning.

solution leads to a fixed upper bound on the number of remote references as well as achieving higher concurrency and lower per-node overhead than the locking alternative.)

Code for a (non-fuzzy) adaptive combining tree barrier with local-only spinning appears in figure 4 (page 9). In general form, it mirrors figure 2. The code to eliminate remote spinning while waiting for notification is more or less straightforward. Rather than set an `occupied` flag, a process uses `fetch_and_store` to set a `visitor` pointer. The atomicity of the operation enables it to determine if another process has already acquired the node, in which case it puts that process's pointer back.[4]

The code required to eliminate per-node locks and to avoid the spins while looking for unvisited parent and sibling nodes is somewhat more subtle. With simple `fetch_and_Φ` instructions we cannot change child and parent pointers in a consistent fashion in one atomic step. We have therefore resorted to an asynchronous approach in which processes may see changes to the tree in different orders. In particular, if a process finds that the parent p of w has already been visited, we allow it to proceed immediately to w's grandparent, even though some other process must of necessity be about to change the pointer from w to p. With sufficient skew in the arrival times of processes, changes to the tree occur in the same order as they do in figure 2. When processes arrive at about the same time, however, the "winner" may follow more than one parent pointer to reach, and visit, the root.

When splicing a sibling o into its grandparent p (see figure 1), we change p's child field first, before changing o's parent field. In between, there is a timing window when a process climbing up through w's sibling may find o and attempt to splice it into its *great*-grandparent. Because the updates to o's parent field are unsynchronized, we must take care to recover in the event that they occur in the incorrect order.[5] `Depth` fields in each node enable us to discover whether the new value of a parent field is an improvement on the old, and to restore the old value if necessary. It is possible for a process to climb up through a node when its parent pointer has just been overwritten with an out-of-date value, and before the better value is restored, but no correctness problems result: the process simply follows more pointers than it would have if it had missed the timing window. At first glance, it would appear that a potentially unbounded number of remote references might be performed while executing the loop to update o's parent field. This would violate our claim of performing only a bounded number of remote references per barrier episode. Fortunately, the number of loop iterations is bounded by $\mathit{depth}(o)-1$, since each iteration sees o's parent link move at least one step closer to the root. Moreover, the probability of this worst case is very low; a single iteration is most likely.

One might suspect that recovery might also be required when updating pointers to children, but in fact these updates are serialized. If w is initially the left child of p, then initially only the process that visits w (call this process X) can change p's left child field. Moreover only process X can cause any node to the left of p (other than w) to point to p as parent, so no other process will

[4] By using a more powerful `compare_and_swap` instruction [7] we could eliminate the need to re-write pointers that are erroneously over-written. (`Compare_and_swap (L, O, N)` compares the value in location `L` to `O` and, if they are equal, replaces `O` with `N`, as a single atomic operation. It returns `true` if it performed the swap, and `false` otherwise.) There is no correctness problem with our code as shown, however; the values of `visitor` fields are not used (except to compare them to `&pseudodata`) until after the barrier is achieved, and all mistakenly overwritten values are restored before that time. We have relied on `fetch_and_store` in our algorithms whenever possible because it is available on a wider variety of machines (including those in our experiments).

[5] `Compare_and_swap` does not help in this case; one can read the pointer to determine whether it is desirable to overwrite it, but even an immediately subsequent `compare_and_swap` may fail because some other process has overwritten the pointer in the interim.

```
shared pseudodata : ^node

type node = record
    visited, notified : Boolean := false, false
    owner : ^node := // address of appropriate f field for leaves,
        // &pseudodata for internal nodes
    bin_left, bin_right, bin_parent : ^node := // tree
    depth, inorder : integer := // as appropriate, in unmodified tree;
        // latter allows us to determine left and right descendancy
    left, right, parent : ^node := bin_left, bin_right, bin_parent

type instance = record
    f : ^node := nil    // node at which we were woken up
    root, my_leaf, my_internal_node : ^node := // as appropriate, in tree
        // my_internal_node is used only for reinitialization

private instances : array [0..2] of instance
    // assumed to lie in memory accessible to other processes
private current_instance : ^instance := &instances[0]
private previous_instance : ^instance := &instances[2]

procedure enter_barrier ()
    n : ^node := current_instance^.my_leaf
    loop
        w : ^node := n^.parent
        if w = nil
            // signal achievement of barrier
            current_instance^.root^.notified := true
            current_instance^.root^.owner^ := current_instance^.root
                // tell owner, if any, that we woke it up at the root
            return
        if fetch_and_store (&n^.visited, true) = false
            exit loop
        n := w

    // adapt tree:
    o : ^node := if n^.inorder < w^.inorder then w^.right else w^.left
    p : ^node := w^.parent
    if p = nil
        o^.parent := nil
    else
        (if w^.inorder < p^inorder then p^.left else p^.right) := o
        loop
            t : ^node := fetch_and_store (&o^.parent, p)
            if t^.depth > p^.depth      // swap was a good thing
                exit loop
            p := t

procedure exit_barrier ()
    n : ^node := current_instance^.my_leaf
    if n^.notified
        goto rtn

    p : ^node := n^.bin_parent
    loop
        if p^.owner = &pseudodata
            p^.owner := &current_instance^.f
            if p^.notified
                exit loop
            else if p^.bin_parent = nil
                repeat
                    p := current_instance^.f
                until p <> nil          // spin
                exit loop
                else
                    p := p^.bin_parent
            else if p^.notified
                exit loop
            else
                repeat
                    p := current_instance^.f
                until p <> nil
                exit loop
        // work way back down to leaf, giving notifications
        while p <> current_instance^.my_leaf
            if n^.inorder < p^.inorder
                o := p^.bin_right
                p := p^.bin_left
            else
                o := p^.bin_left
                p := p^.bin_right
            o^.notified := true
            o^.owner^ := o
    rtn:
        reinitialize (previous_instance)
        previous_instance := current_instance
        current_instance := if current_instance = &instances[2]
            then &instances[0] else current_instance + 1
```

Figure 5: A fuzzy adaptive combining tree barrier with local-only spinning.

acquire the ability to modify *p*'s left child field until after *X* has first made *p* point to *o*, and then made *o* point to *p*. Inorder traversal numbers allow us to determine whether a given node is to the left or the right of its parent without inspecting the parent's (possibly inconsistent) child pointers.

Because of the asynchrony with which processes climb the tree, a slow process can still be modifying pointers when all of its peers have left the barrier and continued other work. We are therefore unable to reinitialize nodes on the way out of the barrier, as did Gupta and Hill in figure 2. Instead, we employ *three* sets of data structures. We reinitialize the one that was used before the current barrier episode, and that will not be used again until after the next episode. Each process takes responsibility for reinitializing its own leaf and one (statically determined) internal node.

3.2. The Fuzzy Variant

To eliminate remote spinning from the code in figure 3, we must again replace `notify` flags with pointers to local flags, eliminate `test_and_set` locks, and adapt the tree asynchronously, just as we did in the non-fuzzy version. Code to enter the barrier and adapt the tree can be taken almost verbatim from figure 4. In `exit_barrier`, however, we must find a way for processes to climb to the highest unoccupied node without the double-locking of the original fuzzy algorithm. In figure 3, a process retains the lock on a child node while locking and inspecting its parent. If the parent is unoccupied, the process releases the child. If the parent is already occupied, the process occupies the child. Our solution is again to adopt an asynchronous approach, in which each process writes a pointer to its wakeup flag into *every* node that appears to be unoccupied on the path from its leaf to the root.

Code for a fuzzy adaptive combining tree barrier with local-only spinning appears in figure 5. With sufficient skew in arrival times, processes will write pointers to their wakeup flags into distinct nodes of the tree, ending at the same nodes at which they would have ended in figure 3. If processes arrive at about the same time, however, more than one of them may write a pointer to its wakeup flag into the same node. Since every process begins by writing its pointer into a leaf, at least one of the pointers will never be overwritten. As an optimization, each process is informed at wakeup time of the node at which its pointer was found, saving it the trouble of performing redundant wakeups at high-level nodes at which its pointer was overwritten.

For the sake of space and time efficiency, we have written `exit_barrier` as a non-recursive routine. As in our non-fuzzy barrier, we employ three sets of data structures to cope with the asynchrony of tree adaptations and reinitialization. By reinitializing the data structures of the previous barrier instance, we eliminate the need for the `last_visitor` fields, which account for much of the complexity of figure 3.

3.3. Breadth-First Wakeup

As noted in section 1, existing logarithmic time barriers (including those above) incorporate synchronization statements that constrain the relative orders in which processes can proceed. This synchronization can necessitate additional context switches when processors are multiprogrammed. In contrast, centralized counter-based barriers permit a process to proceed as soon as all of its peers have arrived, regardless of whether any particular peers have noticed this condition yet or not.

Combining tree barriers avoid static synchronization ordering during the arrival phase of a barrier episode. Unfortunately, the wakeup phase of a barrier episode still embodies static synchronization constraints: processes waiting near the leaves of the tree cannot proceed, even though the barrier has been achieved, until processes higher in the tree take explicit action to awaken them.

In an attempt to combine the best properties of centralized and logarithmic barriers, we have modified the wakeup phase of our local-spinning adaptive combining tree barrier. Specifically, we awaken processes in breadth-first order, abandoning the pursuit of subtrees in which some lower-level process has awoken and is working on our behalf, or in which no process has yet reached the root. The fuzzy version of our code appears in figure 6; the non-fuzzy version is the same, except that `enter_barrier` and `exit_barrier` are concatenated into a single `barrier` routine.

4. Performance Results

We have compared the performance of the various forms of combining tree barrier with that of the centralized and logarithmic barriers found to perform best in previous experiments [14]. After describing our experimental environment in section 4.1, we consider latency for non-fuzzy barriers in section 4.2, focusing in particular on the impact of skew in arrival times. We consider the utility of fuzzy algorithms in section 4.3, with an emphasis on the cross-over points at which the savings in needless spinning makes up for additional overhead. Finally, we consider the issue of multiprogramming in section 4.4, examining both spin-based and event-based synchronization.

4.1. Experimental Environment

Many of our timing tests were performed on a BBN Butterfly 1 multiprocessor. Others were performed on a more modern BBN TC2000 machine at Argonne National Laboratory. The Butterfly 1 consists of up to 256 processor nodes, each of which contains an 8 MHz MC68000 microprocessor and up to four megabytes of memory (one on our machine). There are no caches, coherent or otherwise. Each processor can access its own memory directly, and can access the memory of any node through a $\log_4$-depth switching network. Transactions on the network are packet-switched and non-blocking. If collisions occur at a switch node, one transaction succeeds and all of the others are aborted, to be retried at a later time (in firmware) by the processors that initiated them. In the absence of contention, a remote memory reference (read) takes about 4 μs, roughly 5 times as long as a local reference.

```
shared pseudodata : ^node

type node = record
    visited : Boolean := false
    notified : Boolean := false
    owner : ^node := nil        // who is waiting here?
    bin_left, bin_right, bin_parent : ^node := // tree
    depth : integer := // as appropriate, in unmodified tree
    inorder : integer := // as appropriate, in unmodified tree,
        // to determine left and right descendancy
    left, right, parent : ^node := bin_left, bin_right, bin_parent

type instance = record
    f : ^node := nil    // node at which we were woken up
    root, my_leaf, my_internal_node : ^node := // as appropriate, in tree
        // my_internal_node is used only for reinitialization

private instances : array [0..2] of instance
    // assumed to lie in memory accessible to other processes
private current_instance : ^instance := &instances[0]
private previous_instance : ^instance := &instances[2]

procedure wakeup (p : ^node)
    // Wake up processes under p, in breadth-first order.  Pass off
    // responsibility for a subtree as soon as somebody else is visibly
    // active in it.  Stop working on any subtree whose root is unowned.
    node_queue : queue of ^node
    if leaf (p) return
    loop
        l : ^node := p^.bin_left
        l^.notified := true
        if l^.owner != &pseudodata      // someone is waiting at l
            l^.owner^ := l
            if not leaf (l)
                node_queue.enqueue (l)
        r : ^node := p^.bin_right
        r^.notified := true
        if ^.owner != &pseudodata       // someone is waiting at r
            r^.owner^ := r
            if not leaf (r)
                node_queue.enqueue (r)
        loop
            if node_queue.empty
                return
            p := node_queue.dequeue ()
            if not p^.bin_left^.notified
                exit loop               // nobody else is awake yet at p

procedure enter_barrier ()
    n : ^node := current_instance^.my_leaf
    loop w : ^node := n^.parent
        if w = nil
            // signal achievement of barrier
            current_instance^.root^.visitor^ := current_instance^.root
            wakeup (current_instance^.root)
            return
        x : ^node := fetch_and_store (&w^.visited, true)
        if x = false
            exit loop
        n := w

    // adapt tree:
    o : ^node := if n^.inorder < w^.inorder then w^.right else w^.left
    p : ^node := w^.parent
```

```
        if p = nil
            o^.parent := nil
        else
            (if w^.inorder < p^.inorder then p^.left else p^.right) := o
            loop
                t : ^node := fetch_and_store (&o^.parent, p)
                if t^.depth > p^.depth          // swap was a good thing
                    exit loop
                p := t

    procedure exit_barrier ()
        n : ^node := current_instance^.my_leaf
        if not n^.notified
            // we should move upwards
            p : ^node := n^.bin_parent
            loop
                if (p^.owner = &pseudodata) or ((p^.bin_parent != nil)
                        and (p^.owner = p^.bin_parent^.owner))
                    // we should grab this node
                    p^.owner := &current_instance^.f
                    if p^.notified
                        exit loop
                    if p^.bin_parent = nil
                        repeat
                            p := current_instance^.f
                        until p <> nil          // spin
                        exit loop
                    p := p^.bin_parent
                elsif p^.notified
                    exit loop
                else
                    repeat
                        p := current_instance^.f
                    until p <> nil              // spin
                    exit loop
            wakeup (p)
        reinitialize (previous_instance)
        previous_instance := current_instance
        current_instance := if current_instance = &instances[2]
            then &instances[0] else current_instance + 1
```

Figure 6: A fuzzy adaptive combining tree barrier with local-only spinning and breadth-first process wakeup.

The TC2000 is architecturally similar to the Butterfly 1, but employs 20 MHz MC88100 processors with (non-coherent) caches and a faster $\log_8$-depth switching network based on virtual circuit connections rather than packet switching. With caching disabled, a remote memory reference takes about 1.9 μs, slightly over 3 times as long as a local reference, and about 13 times as long as a cache hit. Experiments by Markatos and LeBlanc [12] indicate that while the TC2000 has relatively good switch bandwidth and latency, it is starved for shared memory bandwidth. One would therefore expect the centralized barriers to perform comparatively badly on the TC2000; our results confirm this expectation.

The Butterfly 1 supports two 16-bit atomic operations: `fetch_and_clear_then_add`, and `fetch_and_clear_then_xor`. Each operation takes three arguments: the address of the 16-bit destination operand, a 16-bit mask, and a 16-bit source operand. The value of the destination operand is `and`ed with the one's complement of the mask, and then `add`ed or `xor`ed with the source operand. The resulting value replaces the original value of the destination operand. The previous value of the destination operand is the return value for the atomic operation. Our code uses only `fetch_and_clear_then_add`. For the locks in figures 2 and 3 we perform a `test_and_set` by specifying a mask of `0xFFFF` and an addend of 1; for the other barriers we perform a `fetch_and_store` by specifying a mask of `0xFFFF` and an addend of the value to be stored. The Butterfly 1 does not support `compare_and_swap`.

In comparison to ordinary loads and stores, atomic operations are relatively expensive on the Butterfly 1; `fetch_and_clear_then_add` takes slightly longer than a call to a null procedure. We shall see that this cost has a significant impact on the relative performance of alternative barrier implementations, since different barriers perform different numbers of atomic operations.

The TC2000 supports the MC88100 XMEM (`fetch_and_store`) instruction in hardware, at essentially the same cost as an ordinary memory reference. It provides additional atomic operations in software, but these must be triggered in kernel mode. They are available to user programs only via kernel calls, and as on the Butterfly 1 are relatively expensive.

The barriers included in our timing tests are listed in figure 7, together with an indication of their line types for subsequent graphs. We have used solid lines for non-fuzzy algorithms, and dotted lines for fuzzy algorithms. When one algorithm has fuzzy and non-fuzzy variants, they share the same tick marks.

The *dissemination* barrier is due to Hensgen, Finkel, and Manber [6]. As mentioned in section 1, it employs $\lceil \log_2 P \rceil$ rounds of synchronization operations in a pattern that resembles a parallel prefix computation: in round k, process i signals process $(i+2^k) \bmod P$. The total number of synchronization operations (remote writes) is $O(P \log P)$ (rather than $O(P)$ as in other logarithmic time barriers) but because the Butterfly is based on a multistage interconnect rather than a central communication bus, $\log P$ of these operations can in general proceed in parallel.

Our previous experiments [14] found the dissemination barrier to be the fastest alternative on the Butterfly 1. The *static tree* barrier was a close runner-up. It has a slightly longer critical path, but less overall communication, and might be preferred to the dissemination barrier when the impact of interconnect contention on other applications is a serious concern. Each process in the static tree barrier is assigned a unique tree node, which is linked into a 4-ary arrival tree by a parent link, and into a binary wakeup tree by a set of child links. Upon arriving at the barrier, each process spins on a local word whose four bytes are set, upon arrival, by the process's children. It then sets a byte in its parent and spins on another local word awaiting notification from its parent. The root process starts a downward wave of notifications when it discovers that all of its children have arrived.

•—•	dissemination
⊟—⊟	static tree
—	central flag with proportional backoff
···	fuzzy central flag with proportional backoff
▵—▵	original adaptive combining tree
▵···▵	fuzzy original adaptive combining tree
×—×	local-spinning adaptive combining tree
×···×	fuzzy local-spinning adaptive combining tree
+—+	local-spinning non-adaptive combining tree
+···+	fuzzy local-spinning non-adaptive combining tree
o—o	local-spininng adaptive combining tree with breadth-first wakeup
o···o	fuzzy local-spinning adaptive combining tree with breadth-first wakeup

Figure 7: Barrier algorithms tested.

entral flag with proportional backoff barrier and its fuzzy variant employ a central d wakeup flag. They pause after an unsuccessful poll of the flag for a period of time al to the number of processes participating in the barrier. Our previous experiments technique to be more effective at reducing contention (and increasing performance) a constant pause or a linear or exponential backoff strategy.

the other barriers were introduced in sections 2 and 3. The *original adaptive combin-* d *fuzzy original adaptive combining tree* are from figures 2 and 3. The *local-spinning ombining tree* and *fuzzy local-spinning adaptive combining tree* are from figures 4 and *al-spinning non-adaptive combining tree* and *fuzzy local-spinning non-adaptive com-* e are from figures 4 and 5, without the twelve line block of code that begins with `ee`. The *fuzzy adaptive combining tree with breadth-first wakeup* is from figure 6; the *ombining tree with breadth-first wakeup* is its obvious non-fuzzy variant.

ch of our timing tests we ran 1000 barrier episodes and calculated the average time per Except in the multiprogramming tests (section 4.4), we placed each process on a dif- cessing node, and ran with timeslicing disabled, to avoid interference from the In many of the tests we introduced delays between barrier episodes, or between `rrier` and `exit_barrier` calls in fuzzy tests. The delays were implemented by round a loop whose execution time was calibrated at 10 μs. In some cases the number s was the same in every process. In other cases we introduced random fluctuations. A ion indicating a delay of, say, 1 ms ± 500 μs indicates that the number of iterations was iformly in the closed interval 50..150. Random numbers were calculated off-line prior . In order to obtain a measure of synchronization cost alone, we subtracted delays and ead from the total measured time in each test.

n-fuzzy) Barrier Latency

8 plots the time required to achieve a barrier on the Butterfly 1 against the number of (and hence processors) participating in the barrier, with no inter-episode or fuzzy esults on the TC2000 (not shown) are qualitatively similar, except that performance of lized barriers is markedly worse, presumably due to contention.

an observe that the explicit locking and non-local spinning of the original adaptive tree barriers imposes a large amount of overhead. We can also see the impact of con- e performance of the centralized barriers degrades markedly as the number of proces- ases, and the curves for the original adaptive combining tree barriers begin to turn ound 30 processors. Comparison of the curves for the adaptive and non-adaptive ver- he local-spinning combining tree barrier indicates that adaptation is a small net loss tested conditions. Similarly, fuzziness is a small net loss in the absence of fuzzy com- The dissemination and static tree barriers perform best by an unequivocal margin.

ure 9 we have introduced an inter-episode delay of 1 ms ± 200 μs. Skew in process es serves to reduce contention for the centralized and original adaptive combining tree nd makes adaptation a small net win. Processes arriving at the barrier early perform enables latecomers to detect the barrier sooner, as predicted by Gupta and Hill. Fuzzi- ll a small net loss. The dissemination and static tree barriers remain the clear winners. the TC2000 (not shown) are qualitatively similar, though again the performance of the d barriers is markedly worse.

valuate the impact of arrival time skew on barrier performance, figure 10 plots time per the Butterfly 1 against the maximum fluctuation in inter-episode delay. All tests

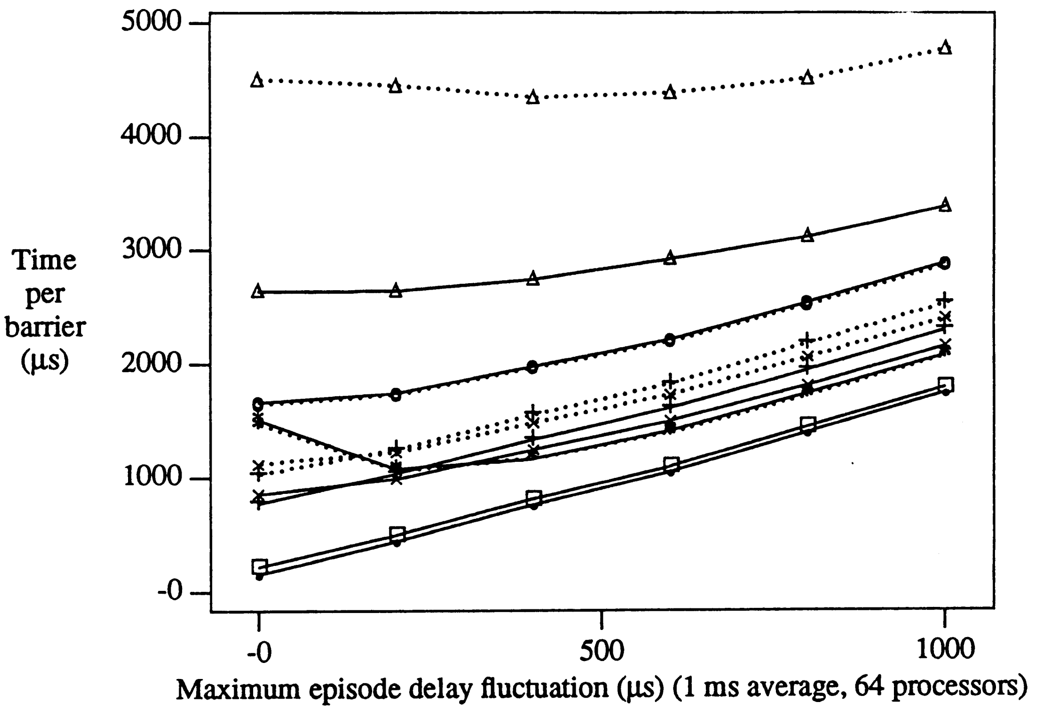

Figure 10: Dependence of performance on skew in process arrival times (Butterfly 1).

employ 64 processes, with inter-episode delays of 1 ms $\pm x$ μs. For the dissemination barrier, the static tree barrier, and the non-adaptive combining tree barriers, synchronization time displays an almost perfectly linear dependence on the expected arrival time of the last arriving process. In each of these algorithms, 1 ms in episode delay fluctuation adds almost exactly 1 ms to synchronization time. Once again, the TC2000 results for non-centralized barriers (not shown) are qualitatively similar.

In the various adaptive combining tree barriers, adaptation serves to mitigate to some extent the increase in synchronization time due to increased fluctuation in episode delay. For the original adaptive combining tree barrier, performance actually improves with small amounts of fluctuation. Overall, the adaptive local-spinning algorithms perform slightly better than their non-adaptive counterparts, though we shall see in section 4.3 that fuzziness negates this advantage. The impact of contention can be seen clearly in the curves for the centralized barriers; their performance improves dramatically with small amounts of skew in process arrival times: minor delays in some processes allow other processes to finish their work and get out of the way.

4.3. Barrier Episodes with Fuzzy Delay

In figures 11 through 14, we have added a fuzzy delay to each iteration of the timing loop. We again plot time per barrier against the number of participating processes (processors). We display results for both the Butterfly 1 and the TC2000. The fuzzy delay is incurred between calls to `enter_barrier` and `exit_barrier` in the case of the fuzzy algorithms, and immediately after the call to `barrier` in the non-fuzzy algorithms (in other words, the fuzzy delay is in addition to the episode delay). The fuzzy delay is 200 μs in figures 11 and 12, and 500 μs in figures 13 and 14. The centralized barriers have been left out of figures 12 and 14; their curves are highly erratic, and high enough at points to significantly shrink the scale of the graph.

In all cases the fuzzy versions of the centralized and local-spinning adaptive combining tree barriers outperform the non-fuzzy versions by significant amounts. With a large fuzzy delay on

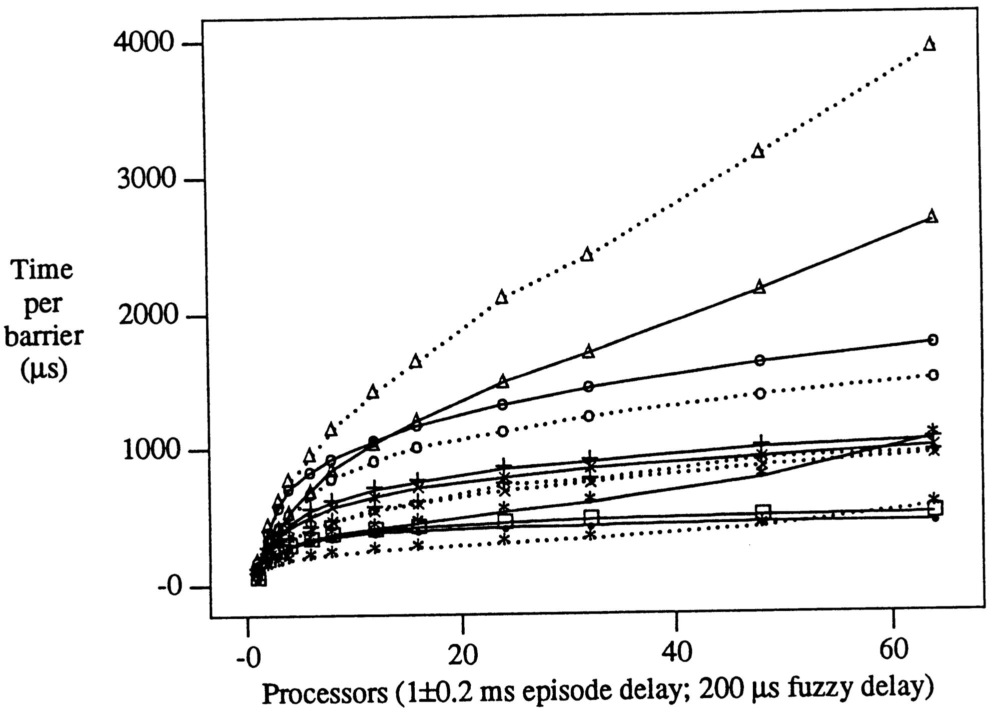

Figure 11: Barrier performance with a modest amount of fuzzy computation (Butterfly 1).

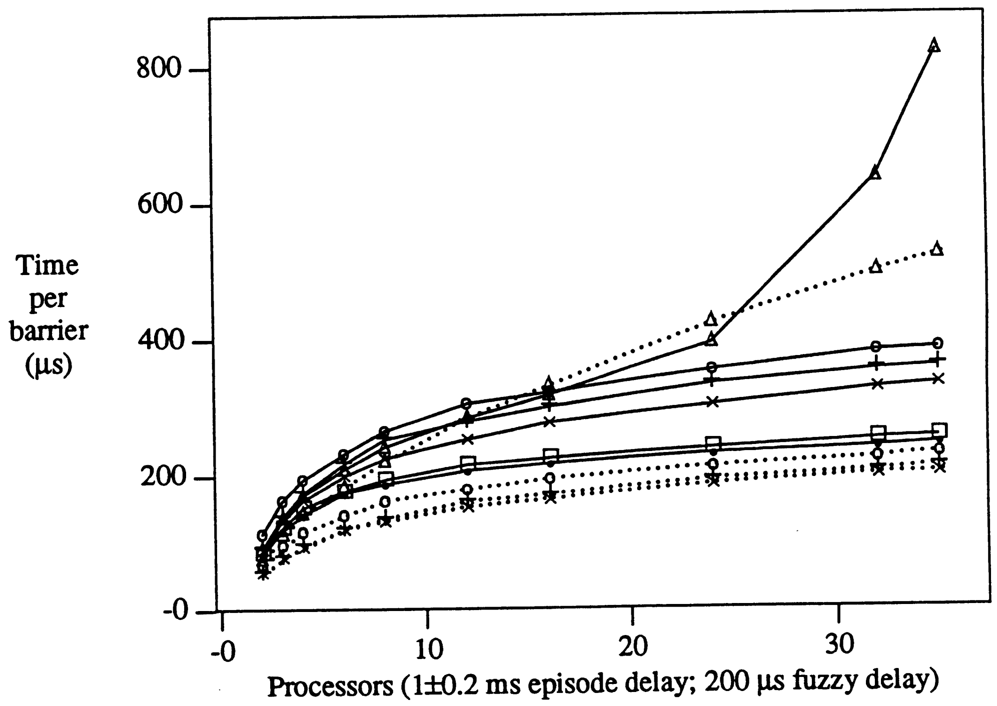

Figure 12: Barrier performance with a modest amount of fuzzy computation (TC2000).

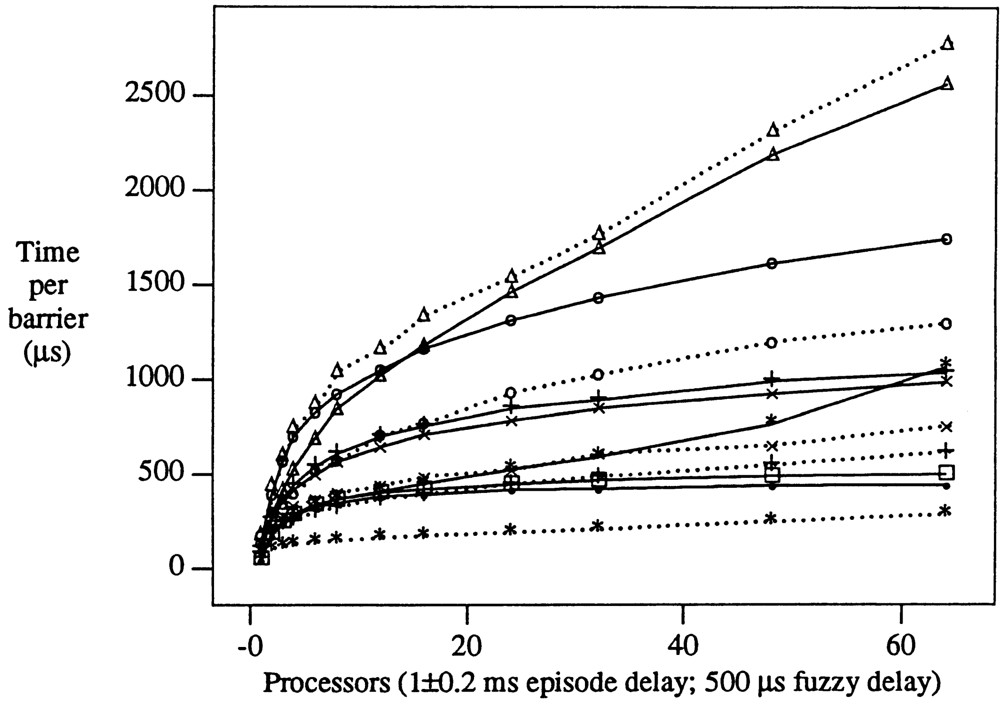

Figure 13: Barrier performance with a large amount of fuzzy computation (Butterfly 1).

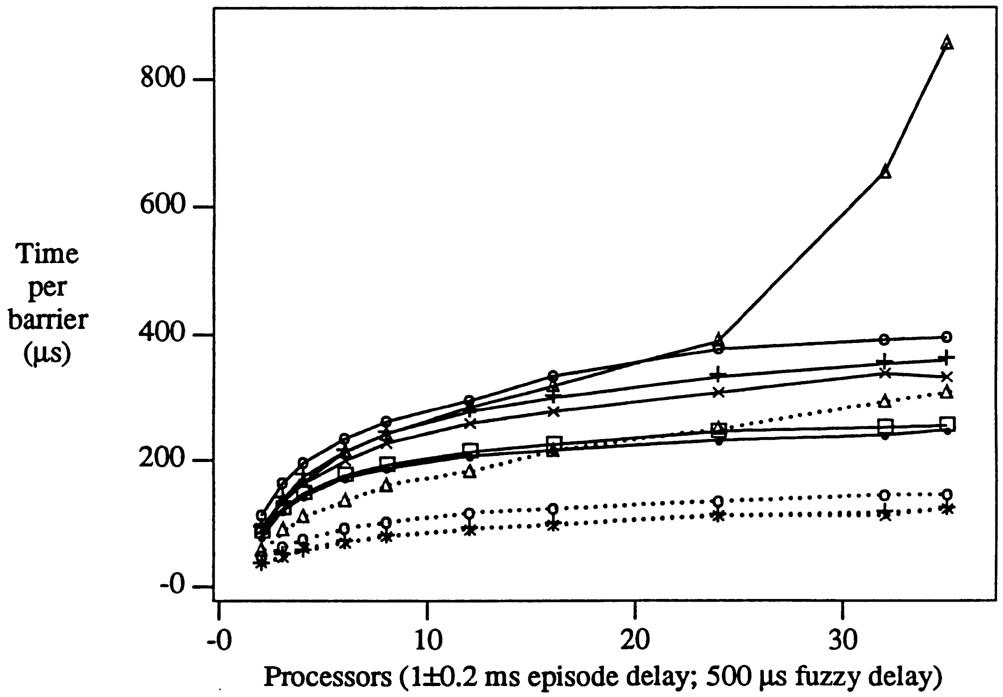

Figure 14: Barrier performance with a large amount of fuzzy computation (TC2000).

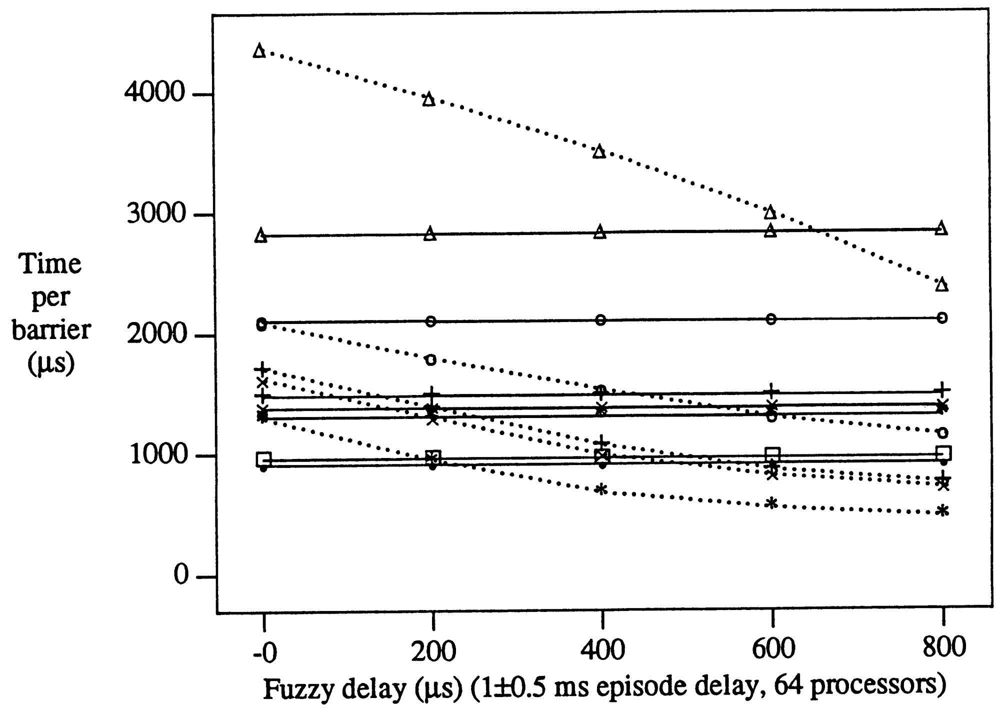

Figure 15: Dependence of performance on amount of fuzzy computation (Butterfly 1).

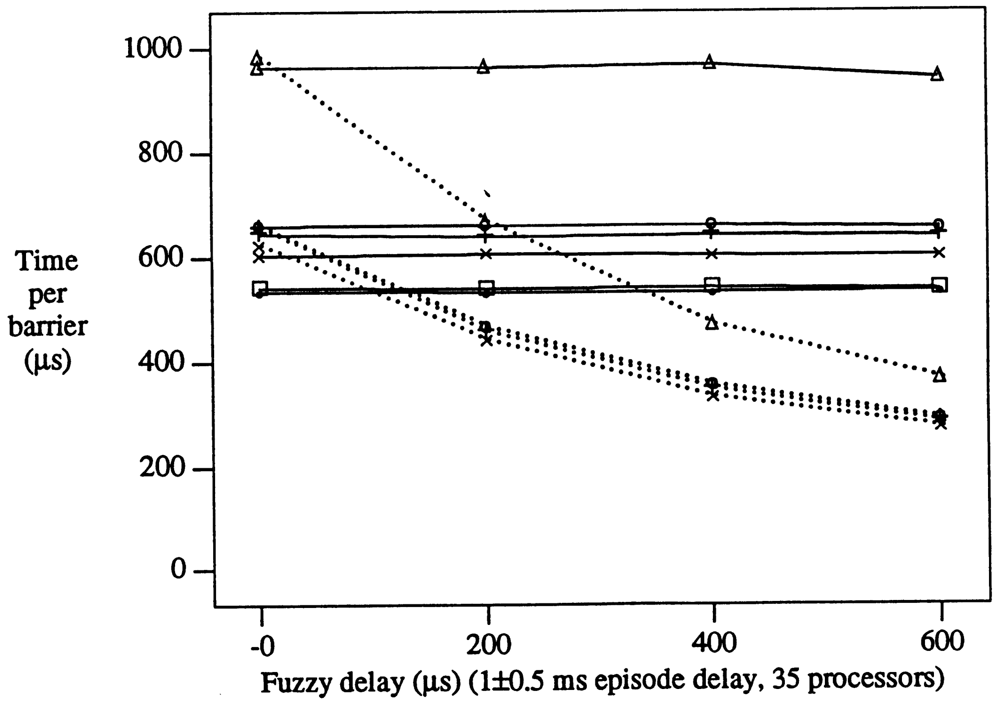

Figure 16: Dependence of performance on amount of fuzzy computation (TC2000).

the Butterfly 1 (figure 13), the margin is large enough to enable the fuzzy centralized barrier to outperform the dissemination and static tree barriers all the way out to 64 processors (though clearly the curves would cross again on a bigger machine). In neither of the Butterfly 1 graphs (figures 11 and 13) does the advantage of fuzziness serve to overcome the overhead of additional locking in the original adaptive combining tree barrier. On the TC2000, fuzziness in the original adaptive combining tree barrier is a clear win with a large fuzzy delay (figure 14), and a modest win for large numbers of processors with a modest fuzzy delay (figure 12).

The additional overhead of fuzziness can be seen directly in figures 15 and 16, for the Butterfly 1 and TC2000, respectively. Here we have plotted the time required for processors to achieve a barrier against the length of the fuzzy delay (in addition to a 1 ± 0.5 ms inter-episode delay). For the centralized and breadth-first wakeup barriers, separating `enter_barrier` from `exit_barrier` introduces no overhead beyond the additional subroutine call. The separation therefore pays off with even very small fuzzy delays. For the adaptive and non-adaptive versions of the local-spinning combining tree barriers, the extra walk up the tree incurs overhead that is recovered almost immediately on the TC2000, and for fuzzy delays starting around 150 μs (15% of the inter-episode delay) on the Butterfly 1. For the original adaptive combining tree barrier, we need almost 700 μs before fuzziness pays off on the Butterfly 1, but less than 100 μs on the TC2000.

Considering the overhead of adaptation, we note that with a modest amount of fuzzy delay the (local-spinning) adaptive combining tree barrier continues to outperform the non-adaptive combining tree barrier by a small amount. With a large fuzzy delay, however, the fuzzy algorithms enable processors to remain busy all the time, obviating the need for adaptation. With a large fuzzy interval (50% of the inter-episode delay) on 64 processors of the Butterfly 1, the fuzzy non-adaptive combining tree barrier outperforms the fuzzy adaptive combining tree barrier by about 20%. On the TC2000, the fuzzy adaptive and non-adaptive combining tree barriers perform about the same.

The crucial observation from these experiments is that on a machine with fast atomic operations (i.e. the TC2000), the fuzzy local-spinning adaptive combining tree barrier outperforms all known alternatives when the amount of fuzzy computation exceeds about 10% of the time between barriers.

4.4. Multiprogramming

In the remaining figures we consider the effect of multiplexing more than one process of the same application on each physical processor. In each case we vary the number of processes per processor from 1 to 5, while keeping the total number of processes fixed at 57. (Memory management hardware on the Butterfly 1 limits us to 57 processes when more than 2 reside on the same processor, or 25 when more than 5 reside on the same processor.) After spinning unsuccessfully for about 30 μs, a process yields the processor voluntarily. It repeats this cycle, spinning briefly and then yielding, each time it comes around the scheduler's ready list, until the condition for which it is waiting becomes true.

We present multiprogrammed barrer results for the Butterfly 1 only. Context switching is supported in firmware under the Butterfly 1's Chrysalis operating system, and is consequently fast. Context switching under the TC2000's nX operating system (a derivative of Unix) is roughly an order of magnitude slower, despite the difference in processor speeds. As a result, busy-wait barriers are impractical on a multiprogrammed TC2000, at least with the native OS. Scheduler-based blocking synchronization should be used instead, with an algorithm in which each process always knows which other(s) to awaken, and in which the number of blocking operations is minimized. The various local-spinning combining tree barriers are best in this regard, with $P-1$ total awaited conditions, $\lceil \log_2 P \rceil$ on the critical path.

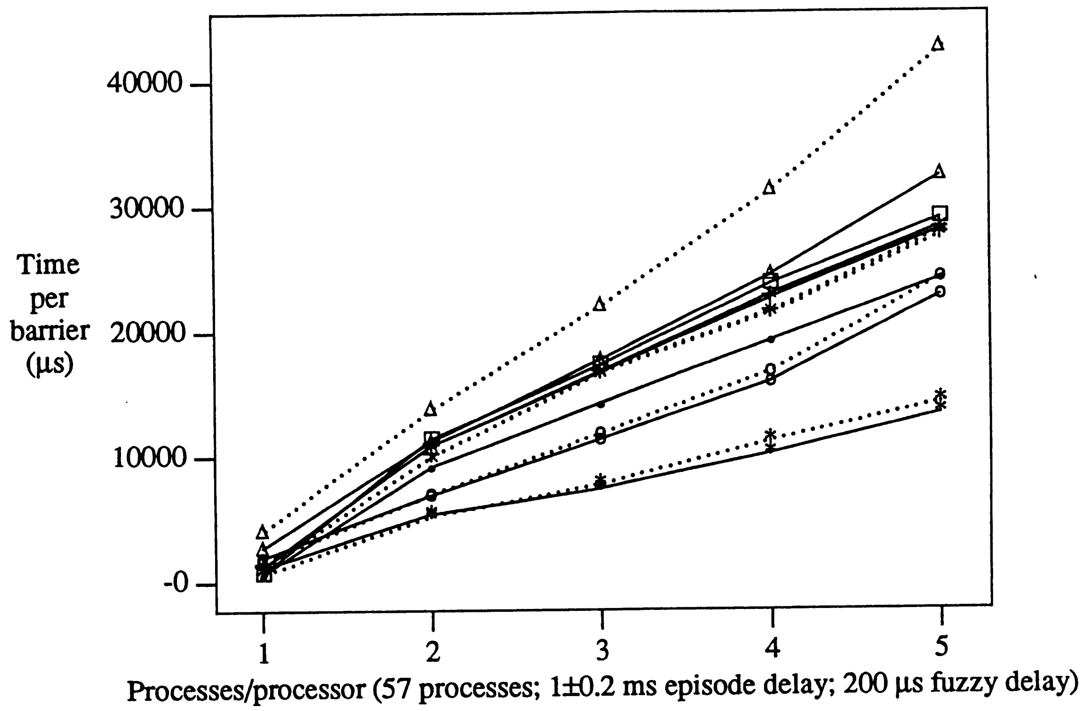

Figure 17: Barrier performance in the presence of multiprogramming.

Figure 17 displays the time required to achieve a multiprogrammed barrier with a 1 ms ± 200 μs inter-episode delay. With one process per processor, the algorithms stack up as shown near the right-hand end of figure 9. (The scale is much smaller in figure 17, of course, and the enabling of timeslicing makes the absolute values slightly worse.) As soon as we introduce the need for context switching, the curves sort out in a different order. The centralized barrier performs well, for two reasons: it embodies no static synchronization constraints, and the contention that is its most serious shortcoming decreases dramatically as increases in the multiprogramming level cause the number of simultaneously active processors to decrease. Among the logarithmic barriers, the dissemination barrier and the adaptive combining tree barrier with breadth-first wakeup outperform the nearest competition by roughly 20%.

The fuzzy versions of the centralized and breadth-first wakeup barriers perform slightly worse than their non-fuzzy counterparts, probably as a result of the extra subroutine call in each process in each barrier episode. The synchronization orders are identical in the fuzzy and non-fuzzy versions of these barriers, and the ability to switch to another process on the same processor when the current process yields essentially negates the advantage of being able to execute code in the same process during the fuzzy interval. The fuzzy versions of the (local-spinning) adaptive and non-adaptive combining tree barriers perform slightly better than their non-fuzzy counterparts. We attribute this effect to differences in synchronization orders. Because of the extra tree traversal in `exit_barrier`, processes in the fuzzy versions of these barriers wake up in roughly the same order they went to sleep. In the non-fuzzy versions, they wake up in roughly reverse order. The simple round-robin scheduling of the Butterfly's Chrysalis operating system causes roughly FIFO wakeup to occur with fewer context switches than roughly LIFO wakeup.

A more detailed understanding of the impact of context switches and round-robin scheduling can be seen in figures 18 and 19. In figure 18 we have plotted the average number of times that each process yields the processor in each barrier episode. In figure 19 we have plotted the average number of times that each process yields the processor *for each distinct condition*. On

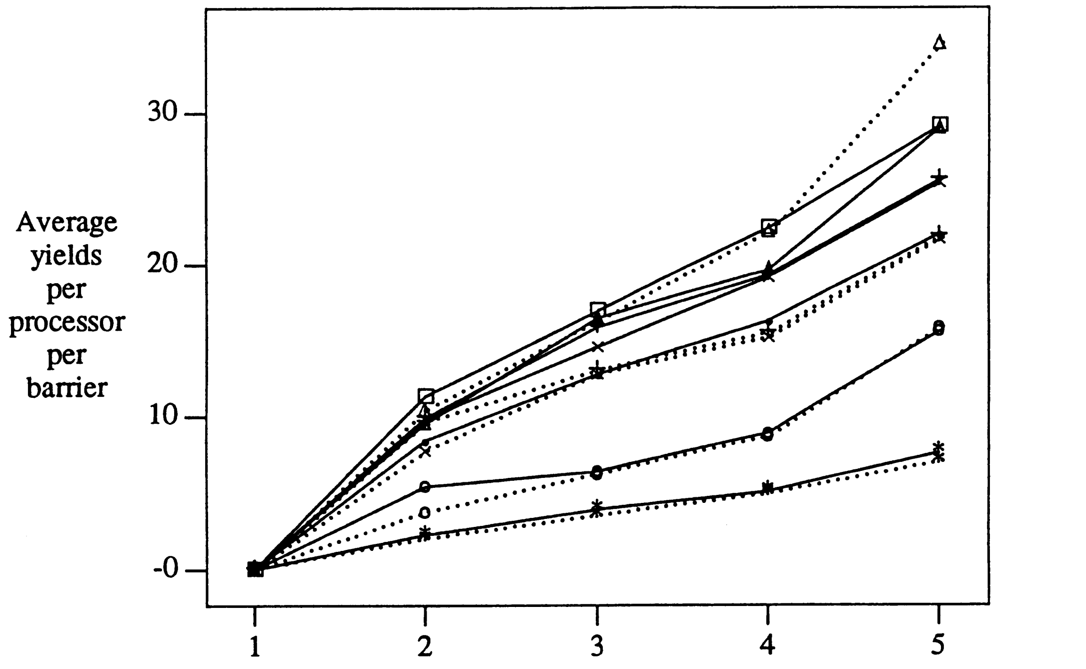

Figure 18: Yields per processor during multiprogrammed barriers.

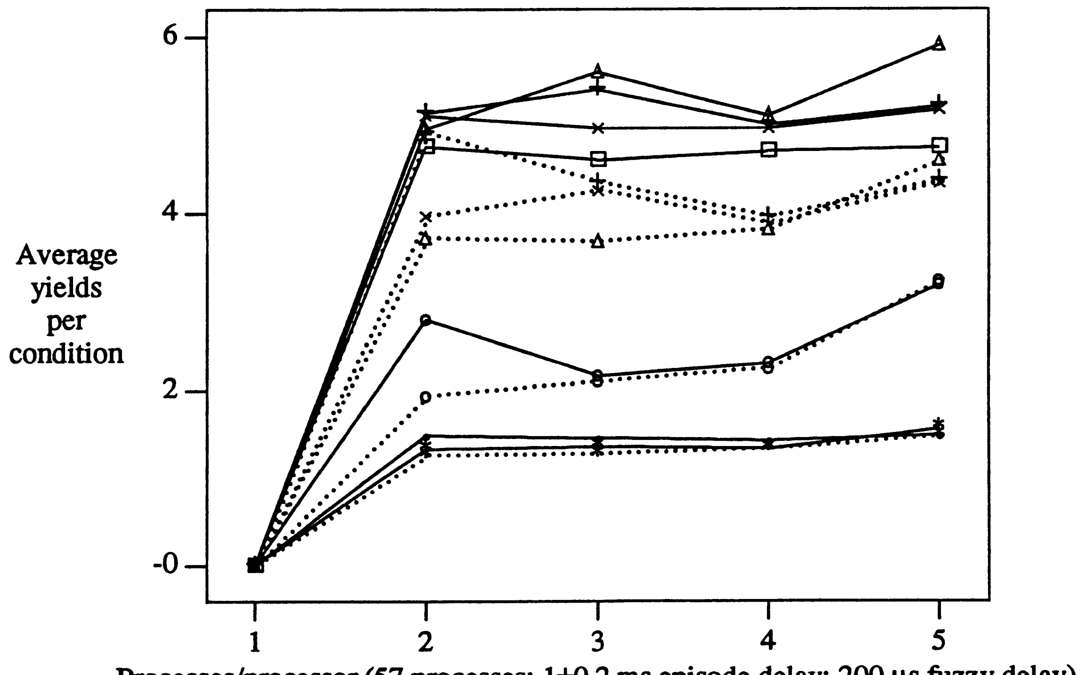

Figure 19: Yields per condition during multiprogrammed barriers.

five processors the dissemination barrier yields the processor for an average of nearly 15 conditions per barrier episode, but in most cases the condition becomes true before a process needs to yield a second time — the average number of yields per condition is less than 1.5. In the original adaptive combining tree barrier, by contrast, a process typically comes around the ready list 5 to 6 times before the condition for which it is waiting is true. The centralized barriers win on both counts on the Butterfly 1; they wait for few distinct conditions, and generally find them true after an average of 1.5 yields. Based on results in the previous section, however, we would not expect the centralized barriers to perform well on the TC2000.

Among the combining tree barriers, the breadth-first wakeup algorithm has a substantial lead in both average yields per condition (2 to 3) and overall context switches per processor per barrier (about 15); it displays the best overall performance. The fact that the fuzzy adaptive and non-adaptive combining tree barriers in figure 19 lie well below their non-fuzzy variants, while the fuzzy centralized and breadth-first wakeup barriers do not, supports our hypothesis regarding round-robin scheduling and LIFO v. FIFO wakeup.

The large number of yields per condition for many of the algorithms in figure 19 suggests the need for scheduler-based synchronization. We explore this possibility in our final experiment, allowing processes to block on kernel-supported event variables, rather than yield the processor and re-test a condition at the start of each following quantum. Blocking on events is an option for barriers in which processes wait for only one condition at a time, and in which the number of processes waiting for the same condition is fixed (and preferably one). The dissemination barrier and the various local-spinning combining tree barriers qualify, but the centralized barriers, the static tree barrier, and the original adaptive combining tree barriers do not.

Figure 20 compares the performance of barrier algorithms that block on events to that of those that spin. The combining tree barriers without breadth-first wakeup show the largest improvement. These are precisely the barriers with the largest number of yields per condition in figure 19. The dissemination barrier and the adaptive combining tree barriers with breadth-first wakeup also show improvement, but to a lesser degree. At the same time, performance with one process per processor is substantially worse when blocking on events. Here the penalty is more than three-fold for the dissemination barrier, which always waits for $\lceil \log_2 P \rceil$ conditions, and more than two-fold for the non-fuzzy combining tree barriers, which wait more often in the absence of multiprogramming than do their fuzzy counterparts.

5. Conclusions

In previous work [14] we concluded:

(1) On a broadcast-based cache-coherent multiprocessor (with unlimited replication), use either a centralized barrier (for modest numbers of processors), or the arrival portion of the static tree barrier, combined with a central wakeup flag.

(2) On a multiprocessor without coherent caches, or with directory-based coherency without broadcast, use either the dissemination barrier or the static tree barrier.

The current study supports these conclusions for machines with only a modest number of processors, or with slow atomic operations. On the Butterfly 1, for example, the dissemination and static tree barriers achieve a substantial advantage over the competition by relying only on ordinary (fast) reads and writes. The advantage is large enough to overshadow the benefits of using barrier algorithms that exploit fuzziness, except perhaps when contention is low enough to make the centralized fuzzy barrier feasible. Even if multiprogramming is a possibility (e.g. because the number of physical processors available to an application may change, while keeping the number of logical processes constant) the combining tree barrier with breadth-first wakeup outperforms the dissemination barrier by relatively small amounts, while the latter outperforms the former by factors of 1.5 to 8 when every process runs on its own processor. If multiprogramming is the

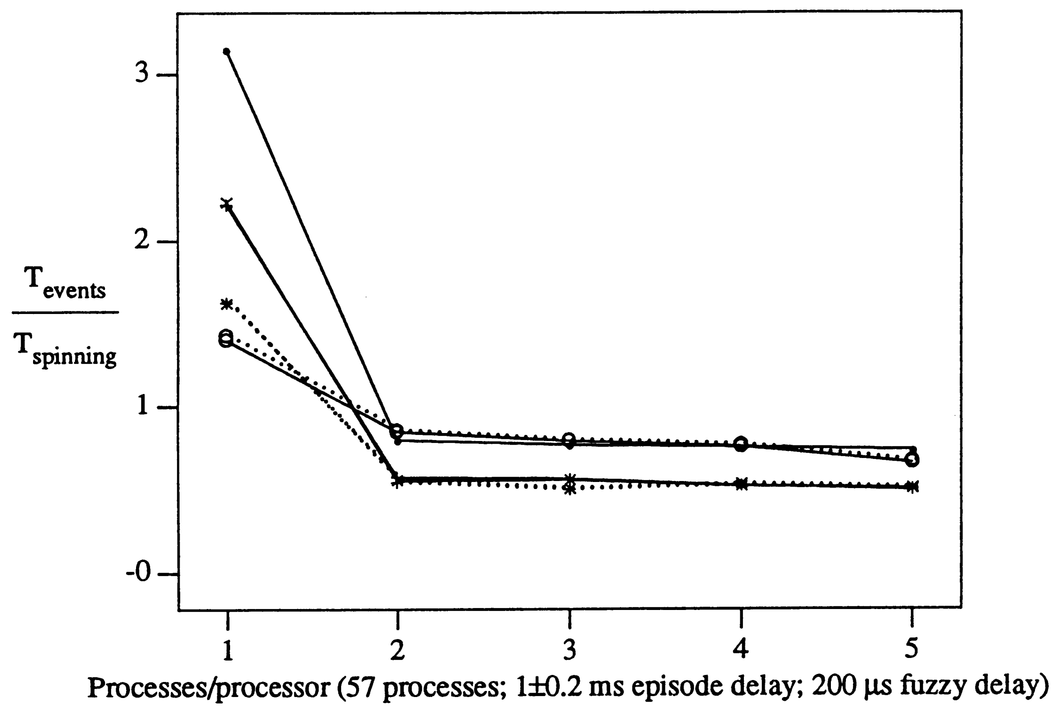

Figure 20: Comparative performance of spinning v. blocking on events.

expected case, then processes should block for events, rather than spinning, probably using one of the local-"spinning" adaptive combining tree barriers.

On the TC2000, the story is quite different. While the dissemination and static tree barriers continue to outperform the various adaptive combining tree barriers in the absence of fuzzy computation, the comparative speed of `fetch_and_store` (XMEM) on the TC2000 enables their fuzzy variants to outperform the more traditional logarithmic barriers when fuzzy computation exceeds roughly 10% of the time between barriers. When fuzzy computation reaches a third of the total time between barriers (see figure 16), the fuzzy local-spinning adaptive combining tree barriers take only half the time of the dissemination and static tree barriers.

A complete set of recommendations appears in figure 21. While all of the experiments in the current study were conducted on machines without coherent caches, the presence of such hardware support would have little effect on the results. The key to fast synchronization is to avoid situations that require inter-processor coherence traffic. The principal advantage of cache coherence, for synchronization purposes, is that it enables a process to spin locally on a variable whose location in main memory is remote. One can achieve essentially the same effect on hardware without coherent caches by using `fetch_and_store` or `compare_and_swap` to replace a flag in an arbitrary location with a pointer to a flag in a local location, on which a process can then spin locally. This technique is a key part of the algorithms in figures 4, 5, and 6, and should permit most synchronization algorithms designed for cache-coherent machines to be adapted to other machines.

Our results confirm the power of Gupta's fuzzy barrier technique; one can exploit fuzziness without contention in a logarithmic barrier, and it clearly pays to do so. The benefits of adaptation are less clear. The local-spinning adaptive combining tree barrier performed slightly better than its non-adaptive counterpart in some cases, but not by much. Adaptation can only save $c \log P$ time, for some small c, and $\log P$ grows very slowly. More to the point, the exploitation

- few processors
  - → use centralized barrier, preferably fuzzy
- many processors
  - uniprogrammed
    - problem can exploit a fuzzy barrier
      - → use fuzzy local-spinning non-adaptive combining tree barrier
    - problem cannot exploit a fuzzy barrier
      - machine provides broadcast-based cache coherence
        - → use static arrival tree with centralized wakeup flag
      - machine is not cache-coherent, or provides directory-based coherence
        - → use static tree barrier or dissemination barrier
  - multiprogrammed
    - multiprogramming is certain to occur, and blocking is possible
      - → use local-spinning combining tree barrier, but block instead of spinning
    - blocking is not possible, but assignment of processes to processors is known and relatively static
      - → use centralized barrier within processors and logarithmic barrier (choice is not crucial) between processors, as suggested by Markatos et al. [11]
    - process assignment is dynamic or unknown, and either multiprogramming is not certain to occur or else blocking is not possible
      - → use non-fuzzy local-spinning non-adaptive combining tree barrier with breadth-first wakeup

Figure 21: Summary of recommendations.

of fuzzy delays negates the benefits of adaptation. There is no point in doing synchronization work early when one can do productive work instead.

Experiments indicate that our modifications to Gupta and Hill's adaptive combining tree barriers are successful at reducing network traffic (thus eliminating contention), and reduce the cost of these algorithms to the point where they become competitive with the best known alternatives. Since improvements in processor performance are likely to outstrip improvements in memory performance for the foreseeable future, and since hardware designers now routinely implement fast atomic instructions, the fuzzy local-spinning combining tree barriers appear to be of serious practical use.

Acknowledgments

Our thanks to Tom LeBlanc and Evangelos Markatos for their comments on this paper.

References

[1] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems 1*:1 (January 1990), pp. 6-16.

[2] E. D. Brooks III, "The Butterfly Barrier," *International Journal of Parallel Programming 15*:4 (1986), pp. 295-307.

[3] G. Graunke and S. Thakkar, ''Synchronization Algorithms for Shared-Memory Multiprocessors,'' *Computer 23*:6 (June 1990), pp. 60-69.

[4] R. Gupta, ''The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors,'' *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 3-6 April 1989, pp. 54-63.

[5] R. Gupta and C. R. Hill, ''A Scalable Implementation of Barrier Synchronization Using An Adaptive Combining Tree,'' *International Journal of Parallel Programming 18*:3 (June 1989), pp. 161-180.

[6] D. Hensgen, R. Finkel and U. Manber, ''Two Algorithms for Barrier Synchronization,'' *International Journal of Parallel Programming 17*:1 (1988), pp. 1-17.

[7] M. Herlihy, ''Wait-Free Synchronization,'' *ACM Transactions on Programming Languages and Systems 13*:1 (January 1991), pp. 124-149.

[8] W. C. Hsieh and W. E. Weihl, ''Scalable Reader-Writer Locks for Parallel Systems,'' MIT/LCS/TR-521, Laboratory for Computer Science, MIT, November 1991.

[9] C. A. Lee, ''Barrier Synchronization over Multistage Interconnection Networks,'' *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, December 1990, pp. 130-133.

[10] B. Lubachevsky, ''Synchronization Barrier and Related Tools for Shared Memory Parallel Programming,'' *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989, pp. II:175-II:179.

[11] E. Markatos, M. Crovella, P. Das, C. Dubnicki and T. LeBlanc, ''The Effects of Multiprogramming on Barrier Synchronization,'' *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, December 1991, pp. 662-669.

[12] E. P. Markatos and T. J. LeBlanc, ''Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance,'' TR 420, Computer Science Department, University of Rochester, March 1992.

[13] J. M. Mellor-Crummey and M. L. Scott, ''Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors,'' *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, 21-24 April 1991, pp. 106-113.

[14] J. M. Mellor-Crummey and M. L. Scott, ''Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,'' *ACM Transactions on Computer Systems 9*:1 (February 1991), pp. 21-65.

[15] J. M. Mellor-Crummey and M. L. Scott, ''Synchronization Without Contention,'' *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 8-11 April 1991, pp. 269-278. In *ACM SIGARCH Computer Architecture News 19*:2, *ACM SIGOPS Operating Systems Review 25* (special issue), and *ACM SIGPLAN Notices 26*:4.

[16] P. Yew, N. Tzeng and D. H. Lawrie, ''Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,'' *IEEE Transactions on Computers C-36*:4 (April 1987), pp. 388-395.