Scalable Concurrent Counting

Maurice Herlihy * Beng-Hong Lim † Nir Shavit ‡

August 12, 1994

Abstract

The notion of *counting* is central to a number of basic multiprocessor coordination problems, such as dynamic load balancing, barrier synchronization, and concurrent data structure design. In this paper, we investigate the scalability of a variety of counting techniques for large-scale multiprocessors. We compare counting techniques based on: (1) spin locks, (2) message passing, (3) distributed queues, (4) software combining trees, and (5) counting networks. Our comparison is based on a series of simple benchmarks on a simulated 64-processor Alewife machine, a distributed-memory multiprocessor currently under development at MIT. Although locking techniques are known to perform well on small-scale, bus-based multiprocessors, serialization limits performance and contention can degrade performance. Both counting networks and combining trees substantially outperform the other methods by avoiding serialization and alleviating contention, although combining tree throughput is more sensitive to variations in load. A comparison of shared-memory and message-passing implementations of counting networks and combining trees shows that message-passing implementations have substantially higher throughput.

*Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA 02139
†Laboratory for Computer Science, MIT, Cambridge, MA 02139
‡Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel 69978

A preliminary version of this report appeared in the Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, July 1992, San Diego, CA [16].

1 Introduction

The notion of *counting* is central to a number of basic multiprocessor coordination problems, such as dynamic load balancing, barrier synchronization, and concurrent data structure design. (See Freudenthal and Gottlieb [11] for further examples.) For our purposes, a *counter* is an object that holds an integer value, and provides a *fetch-and-increment* operation that increments the counter's value and returns its previous value. The values returned may represent addresses in memory, loop or array indices, program counters, or destinations on an interconnection network.

It is difficult to design software counting techniques that scale well. The challenge is how to ensure that the counter's throughput continues to increase as the level of concurrency increases. There are two reasons why it is difficult for throughput to keep up with concurrency: contention in memory and interconnect, and unwanted serialization (i.e., absence of parallelism). In this paper, we present the results of an experimental investigation of the scalability of a variety of software counting techniques. We consider five basic techniques:

1. Lock-based counters, encompassing both test-and-test-and-set [21] locks with exponential backoff [1, 4, 15], and a version of the MCS queue lock that relies only on atomic swaps [19].

2. A message-based counter, in which a single processor increments the counter in response to messages.

3. A queue-based counter, which is a version of the MCS queue lock [19] optimized for distributed counting.

4. Software combining trees [12, 25].

5. Counting networks [5].

For each technique, we ran a series of simple benchmarks on a simulated 64-processor Alewife machine [2], a cache-coherent distributed-memory machine currently under development at MIT. Our experiments were done on the ASIM simulator, an accurate cycle-by-cycle simulator for the Alewife architecture. ASIM is the principal simulator used by the Alewife research group.

Each of the techniques we consider has been independently proposed as a way to perform scalable synchronization in highly concurrent systems.

Here, for the first time, they are compared directly on a realistic large-scale shared-memory multiprocessor.

Our results suggest the following:

- For a concurrent counting technique to be scalable, it must have two distinct properties. First, it must avoid generating high levels of memory or interconnect contention, and second, it must permit concurrent increment operations to proceed in parallel.

- For some techniques, such as the lock-based counters, contention causes performance to degrade substantially at higher levels of concurrency. Earlier experimental work on small-scale multiprocessors has shown that spin locks with exponential backoff and queue locks both perform well for certain kinds of problems on bus-based architectures [4, 15, 19]. Nevertheless, our results indicate that these techniques do not scale well to large-scale distributed memory multiprocessors. As concurrency increases, both spin locks with exponential backoff and queue locks are severely affected by contention.

- Other techniques, such as the message and queue-based counters, are relatively impervious to contention, but nevertheless scale poorly because the absence of concurrency causes throughput to plateau at a relatively low level.

- Software combining trees and counting networks are the only techniques we found to be truly scalable. For both techniques, throughput increases with concurrency for as far as our experiments were able to measure. These techniques avoid contention in the same way: by distributing synchronization operations across a data structure. They support concurrency in different ways: combining trees merge increment requests, while counting networks allow multiple threads to traverse the network at the same time.

- Although both counting networks and software combining trees have similar scaling behavior, combining trees are more susceptible to variations in the inter-arrival times of increment requests because two requests arriving at a node must arrive within a small time window for combining to occur. Additionally, locks that are held for a significant amount of time at the combining tree nodes may block progress up the tree.

- Combining trees and counting networks can be implemented either in distributed shared memory, or directly by message passing and interprocessor interrupts. For both combining trees and counting networks, message passing substantially outperforms shared memory.

We note that the combining tree can compute a general Fetch-and-Φ operation. However, unlike counting networks, it is not lock-free: a stalled process can inhibit other processes from making forward progress. In this respect, counting networks have a substantial advantage over combining trees in systems where individual processes might incur arbitrary delays, an important property for concurrent data structure design.

A preliminary version of some of these results appeared in [16]. This paper extends the earlier paper in the following ways.

- We revise the queue-lock-based counter to use the MCS queue lock instead of the Anderson queue lock [4].
- We add an analysis of a centralized message-based counter.
- We add message-passing implementations of combining trees and counting networks, which we have found to be the most scalable of all the techniques considered.
- We show the importance of parallelism for scalable performance of shared data structures. We do so by comparing two distributed data structures: a counting network and a linearizable counting network. The latter can compute a general Fetch-and-Φ but introduces a sequential waiting chain.
- We present statistics on the combining rates for the software combining tree.

2 Techniques for Concurrent Counting

Table 1 summarizes the five techniques we consider for shared counting. It is convenient to classify these techniques as either centralized or distributed, and as either sequential or parallel. A counter is *centralized* if its value is kept in a unique memory location, and *distributed* if it is kept across a distributed data structure. Access to the counter is *sequential* if requests must update the counter in a one-at-a-time order, and *parallel* if multiple requests can update the counter simultaneously.

Method	Centralized or Distributed	Sequential or Parallel
Lock-based counter	Centralized	Sequential
Message-based counter	Centralized	Sequential
Queue-based counter	Distributed	Sequential
Combining Tree	Centralized	Parallel
Counting Network	Distributed	Parallel

Table 1: Techniques for concurrent counting

Lock-based counter In this technique, the counter is represented by a shared memory location protected by a spin lock. To increment the counter, a processor must acquire the lock, read and increment the memory location, and release the lock. We consider two spin lock algorithms: test-and-test-and-set with exponential backoff [4, 15], and a version of the MCS queue lock that relies only on atomic swaps [19].

Message-based counter In this technique, the shared counter is represented by a private memory location owned by a unique processor. To increment the counter, a processor sends a request message to that unique processor and waits for a reply. The processor receiving the request message increments the counter and sends a reply message containing the value of the counter. Request messages are handled atomically with respect to other request messages.

Queue-based counter This technique is based on the MCS queue lock algorithm, adapted for counting on a network-based multiprocessor. The MCS queue lock maintains a pointer to the tail of a software queue of lock waiters. The lock is free if it points to an empty queue, and is busy otherwise. The process at the head of the queue owns the lock, and each process on the queue has a pointer to its successor. To acquire a lock, a process appends itself to the tail of the queue. If the queue was empty, the process owns the lock; otherwise it waits for a signal from its predecessor. To release a lock, a process checks to see if it has a waiting successor. If so, it signals that

successor, otherwise it empties the queue. See [19] for further details.

The queue-based counter improves on a simple lock-based counter in the following way. Instead of keeping the counter value in a fixed memory location, it is kept at the processor that currently holds the lock. On releasing the lock, that processor passes ownership of the lock and the counter value directly to the next processor in the queue. If there is no next processor, the current value is stored in the lock. This technique combines synchronization with data transfer and reduces communication requirements. Figure 1 shows the pseudocode for this counter following the style of [19].

Software combining tree In a *combining tree*, increment requests enter at a leaf of the tree. When two requests simultaneously arrive at the same node, they are *combined*; one process advances up the tree with the combined request, while the other waits for the result. The combined requests are applied to the counter when they reach the root, and the results are sent back down the tree and distributed to the waiting processes. Hardware combining trees were first proposed as a feature of the NYU Ultracomputer [13].

For our experiments, we implemented the software combining tree algorithm proposed by Goodman et al. in [12]. This algorithm can compute a general Fetch-and-Φ operation, although we use it for the special case of Fetch-and-Increment. A drawback of the algorithm (especially with respect to the counting network algorithm to be presented below) is that delays incurred even by a single process in traversing the tree can inhibit the progress of all others.

Our code for this algorithm is shown in Figures 2 and 3. Because Alewife does not have a `QOSB` primitive, we have omitted all calls to `QOSB`. We also mark in comments a change to enhance performance of the algorithm on Alewife, and a fix to a bug in the original code. (The reader is referred to the original paper [12] for a more complete description of the algorithm.) An earlier software combining tree algorithm proposed by Yew et al. [25] is not suitable for implementing a shared counter because it disallows asynchronous combining of requests.

We investigated two ways to implement combining trees. In a *shared-memory* implementation, each tree node is represented as a data structure in shared memory. Simple *test-and-set* locks are used for atomically updating the nodes. In a *message-passing* implementation, each tree node is private to an individual processor that provides access to the node via message-passing.

```
type qnode = record
  next  : ^qnode
  value : int | nil

type counter = record
  qnode : ^qnode                    // initially nil
  value : int

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in  locally-accessible shared memory
procedure fetch_and_add(C : ^counter, I : ^qnode, v : int) returns int
    value : int := acquire_value(C, I)
    release_value(C, I, value+v)
    return value

procedure acquire_value(C : ^counter, I : ^qnode) returns int
    I->next := nil
    predecessor : ^qnode := fetch_and_store(&C->qnode, I)
    if predecessor != nil
        I->value := nil
        predecessor->next := I       // queue self and
        repeat while I->value = nil  // wait for the value
        return I->value
    else
        return C->value

procedure release_value(C : ^counter, I : ^qnode, value : int)
    if I->next = nil
        C->value := value
        old_tail : ^qnode := fetch_and_store(&C->qnode, nil)
        if old_tail = I return
        usurper : ^qnode := fetch_and_store(&C->qnode, old_tail)
        repeat while I->next = nil
        if usurper != nil
            usurper->next := I->next
        else
            I->next->value := value
    else
        I->next->value := value
```

Figure 1: The MCS-queue-based counter

```
function fetch_and_add(counter : tree, incr : int) returns int

  // Part One
  last_level, saved_result : int
  node : tree_node

  level : int := FIRST_LEVEL
  going_up : boolean := TRUE
  repeat
      node := get_node(counter, level, pid)
      lock(node)
      if node.status = RESULT then
          unlock(node)
          repeat while node.status = RESULT  // change: minimize locking
      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
  while going_up

  // Part Two
  total : int := incr
  level := FIRST_LEVEL
  repeat
      visited : tree_node := get_node(counter, level, pid)
      lock(visited)
      visited.first_incr := total
      if visited.wait_flag then
         total := total + visited.second_incr
      level := level + 1
  while level < last_level
```

Figure 2: Combining Tree Code: Parts One and Two

```
// Part Three
if node.status = COMBINE then
    node.second_incr := total
    node.wait_flag := TRUE
    repeat
        unlock(node)
        repeat while node.status = COMBINE  // change: minimize locking
        lock (node)
    while node.status = COMBINE
    node.wait_flag := FALSE
    node.status := FREE
    saved_result := node.result
else
    saved_result := node.result
    node.result := node.result + total
unlock(node)

// Part Four
level := last_level - 1
repeat
    visited : tree_node := get_node(counter, level, pid)
    if visited.wait_flag then
        visited.status := RESULT
        visited.result := saved_result + visited.first_incr
    else
        visited.status := FREE
    unlock(visited)  // bug fix: need an unlock here
    level := level - 1
while level >= FIRST_LEVEL

return saved_result
```

Figure 3: Combining Tree Code: Parts Three and Four

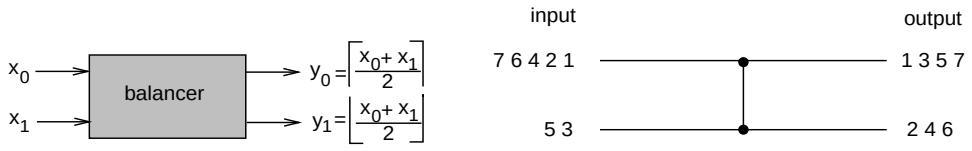

Figure 4: A Balancer.

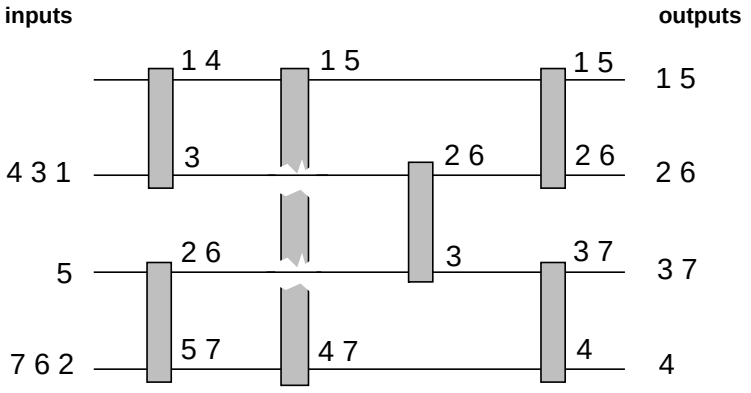

Figure 5: A sequential execution of an input sequence to a BITONIC[4] network.

A Fetch-and-Φ traverses the tree as a series of relayed messages.

Counting network A *counting network* [5] is a highly concurrent data structure used to implement a counter. An abstract counting network, like a sorting network [9], is a directed graph whose nodes are simple computing elements called *balancers*, and whose edges are called *wires*. Each *token* (input item) enters on one of the network's $w \leq n$ input wires, traverses a sequence of balancers, and leaves on an output wire. Unlike a sorting network, a w input counting network can count any number $N \gg w$ of input tokens even if they arrive at arbitrary times, are distributed unevenly among the input wires, and propagate through the network asynchronously.

```
type balancer = record
  type  : [INTERNAL | OUTPUT]
  up    : ^balancer
  down  : ^balancer
  state : boolean               // initially 0
  count : int
  lock  : ^lock

// parameter B, below, points to an input balancer
// of a counting network
procedure traverse_cnet(B : ^balancer)
    next : ^balancer := B
    repeat
        lock(next->lock)
        next->state := 1 - next->state
        unlock(next->lock)
        if state := 0
            next := next->up
        else
            next := next->down
    while next->type != OUTPUT
    lock(next->lock)
    count : int := next->count
    next->count := count + WIDTH
    unlock(next->lock)
    return count
```

Figure 6: Code for traversing a counting network using shared memory operations.

For example, Figure 5 shows a four-input four-output counting network. Intuitively, a balancer (see Figure 4) is just a toggle mechanism that repeatedly forwards tokens to alternating output wires. Figure 5 shows an example computation in which input tokens traverse the network sequentially, one after the other. For notational convenience, tokens are labeled in arrival order, although these numbers are *not* used by the network. In this network, the first input (numbered 1) enters on wire 2 and leaves on wire 1, the second leaves on wire 2, and so on. (The reader is encouraged to try this for him/herself.) Thus, if on the i-th output wire the network assigns to consecutive output tokens the values $i, i+4, i+2\cdot 4, \ldots$, it is *counting* the number of input tokens without ever passing them all through a shared computing element.

Just as for combining trees, we investigated two ways to implement counting networks in software.

- *shared memory:* Each balancer is implemented as a binary variable in shared memory. The value of the variable indicates the output wire on which the next token will exit. The network wiring is kept in tables local to each process. Each process "shepherds" a token through the network by traversing balancers, one after the other, applying an atomic complement operation to determine which balancer to visit next. The atomic complement is implemented in software using simple *test-and-set* locks as in the combining tree implementation. An atomic bit-complement operation would allow a lock-free implementation. The code for traversing a network is shown in Figure 6.

- *message passing:* Each balancer is implemented by variables private to a particular processor. Balancers are assigned to processors at random with a uniform distribution [1]. For balancers internal to the network, two variables name the processors representing the destination balancers of the output wires, and the third, binary variable indicates on which of the two output wires the next token will exit. For output balancers, the two variables hold counter values, and the the third, binary variable indicates which counter will be advanced by the next arriving token. A token is a message that carries the identity of the requesting processor. A process sends a token message to an input balancer, which complements its binary variable and forwards the token. When

[1] Communication delays in Alewife are such that it is not worthwhile trying to place nearby balancers on nearby processors in a 64-processor configuration.

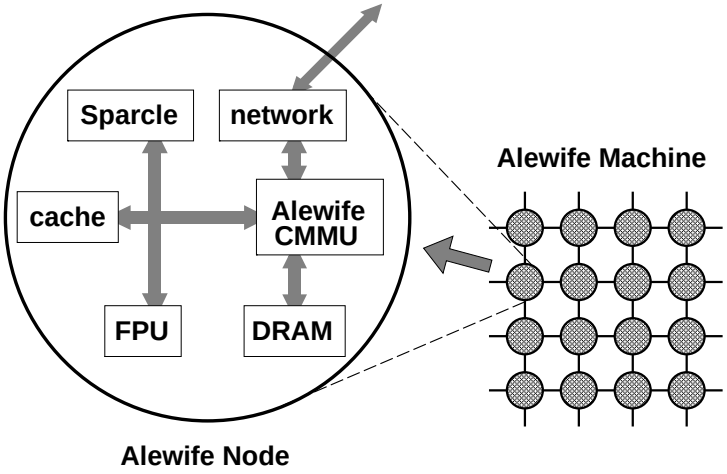

Figure 7: *An Alewife node.*

the token reaches an output balancer, the processor implementing the balancer complements its binary variable, advances the appropriate counter, and sends the result to the original requester.

Counting networks achieve a high level of throughput by decomposing interactions among processors into pieces that can be performed in parallel, effectively reducing memory contention. Aspnes, Herlihy, and Shavit [5] give two $O(\log^2 n)$ depth counting networks. In this paper, we use their *Bitonic* counting network, whose layout is isomorphic to the Bitonic sorting network of Batcher [6]. Henceforth, we use "counting network" to mean "Bitonic counting network."

3 Experimental Methodology

The MIT Alewife multiprocessor [2] is a cache-coherent, distributed-memory multiprocessor that supports the shared-memory programming abstraction. Figure 7 illustrates the high-level organization of an Alewife node. Each node consists of a Sparcle processor [3], an FPU, 64KB of cache memory, a 4MB portion of globally-addressable memory, the Caltech MRC network router, and the Alewife Communications and Memory Management Unit (CMMU) [17].

The CMMU implements a cache-coherent globally-shared address space

with the LimitLESS cache-coherence protocol [8]. The LimitLESS cache-coherence protocol maintains a small, fixed number of directory pointers in hardware, and relies on software trap handlers to handle cache-coherence actions when the number of read copies of a cache block exceeds the limited number of hardware directory pointers. The current implementation of the Alewife CMMU has 5 hardware directory pointers per cache line.

The CMMU also interfaces the Sparcle processor to the interconnection network, allowing the use of an efficient message-passing interface for communication [18]. The LimitLESS protocol relies on this interface to handle coherence operations in software. The message interface also allows us to use message-passing operations to implement the synchronization operations. An incoming message traps the processor and invokes a user-defined message handler. The message handler can be atomic with respect to other message handlers in the style of Active Messages [10].

Our experiments were done on the ASIM simulator, an accurate cycle-by-cycle simulator for the Alewife architecture. This is the principal simulator used by the Alewife research group. In this section, we describe the three synthetic benchmarks we use to compare counting techniques.

3.1 Counting Benchmark

In this benchmark (Figure 8), each processor executes a loop that increments a counter as fast as it can. We measure the number of satisfied increment requests during the interval when all threads are actively issuing requests, and divide that by the length of the interval. From these measurements we arrive at the average throughput of increment requests. This is the simplest possible benchmark, producing the highest levels of concurrency and contention.

3.2 Index Distribution Benchmark

Index distribution is a load balancing technique in which processes dynamically choose independent loop iterations to execute in parallel. (As a simple example of index distribution, consider the problem of rendering the Mandelbrot Set. Each loop iteration covers a rectangle in the screen. Because rectangles are independent of one another, they can be rendered in parallel, but because some rectangles take unpredictably longer than others, dynamic load balancing is important for performance.) A similar application is a *software instruction counter* [20].

```
procedure do_counting(C : ^counter, iters : int)
   i : int := 0
   repeat
     fetch_and_increment(counter)
     i := i + 1
   while (i < iters)
```

Figure 8: Counting Benchmark

```
procedure do_index(C : ^counter, iters : int, w : int)
   repeat
     i := fetch_and_increment(counter)
     delay(random() mod w)
   while (i < iters)
```

Figure 9: Index Distribution Benchmark

In this benchmark (Figure 9), n processes execute 2048 increments, where n ranges from 1 to 64. Each process executes on one processor. Between each increment, each process pauses for a duration randomly chosen from a uniform distribution between 0 and w, where w is 100, 1000, and 5000. The increment models a process taking an index, and the random pause represents the execution of the loop iteration for that index. This benchmark is similar to Bershad's benchmark for lock-free synchronization [7].

3.3 Job Queue Benchmark

A *job queue* is a load balancing technique in which processes dynamically insert and remove jobs from a shared queue. Each process alternates dequeuing a job, working on the job for some duration, and enqueuing a job. The queue itself consists of an array with a flag on each element that signifies if the element is present or not. We use full/empty bits [22] on Alewife to implement this flag. A *head* counter indicates the first full element, and a *tail* counter indicating the first empty element. The elements of the array are distributed across the machine.

A process dequeues an item by incrementing the head counter, and atomically removing one job from the corresponding array position. Enqueues are performed analogously. Note that multiple enqueue and dequeue operations can proceed concurrently, since enqueues synchronize by incrementing the head counter, and dequeues synchronize by incrementing the tail counter.

This benchmark (Figure 10) is structured as follows. For We vary the number of processes, P from 1 to 64. Each process, executing on one processor, repeatedly

1. obtains an index, m, from the head counter
2. dequeues a job from location m modulo P of an array of size P
3. pauses for a duration randomly chosen from a uniform distribution between 0 and w, where w is 100, 1000, and 5000, and then
4. obtains an index, n, from the tail counter
5. enqueues a new job at location n modulo P of the array of size P

The benchmark halts when a total of 2048 jobs have been dequeued and executed by all the processes.

```
type q_elem = record
   value     : int
   not_empty : boolean          // initially 0

job_array : distributed array[0 : P-1] of q_elem

procedure do_job_queue(enq : ^counter, deq : ^counter, njobs : int)
   enq_index, deq_index : int
   repeat
     enq_index := fetch_and_increment(enq_counter)
     enq_job(enq_index mod P, generate_job())
     deq_index := fetch_and_increment(deq_counter)
     job := deq_job(deq_index mod P)
     delay(random() mod w)
   while (deq_index < njobs)

procedure enq_job(index : int, the_job : job)
   repeat while (job_array[index].not_empty)
   job_array[index].value := the_job
   job_array[index].not_empty := true

procedure deq_job(index : int) returns job
   repeat until (job_array[index].not_empty)
   the_job : job  := job_array[index].value
   job_array[index].not_empty := false
   return job
```

Figure 10: Job Queue benchmark

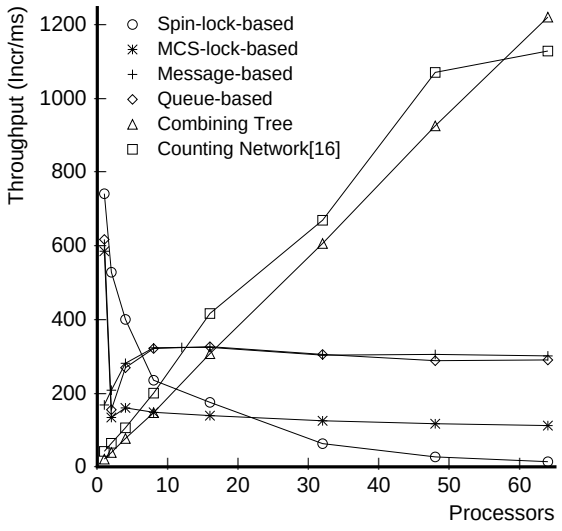

Figure 11: Comparing the throughput of the five counting techniques.

4 Experimental Results

In this section, we present the results of running the benchmarks on various implementations of shared counters on the Alewife simulator. All timings assume a 33 MHz processor clock. In all experiments, we use a radix-2 combining tree with 64 leaves and a counting network of width 16, unless otherwise stated. We first present the results for the counting benchmark. This benchmark gives a sense of the scalability and the peak throughput of each of the counters at different levels of concurrency. We then present the throughput results for the index distribution and job queue benchmarks, which illustrate how the counters would perform under more realistic workloads.

4.1 Counting Benchmark

Figure 11 presents the throughput attained by each of the counting algorithms. We measure the throughput during the interval when *all* processors are actively incrementing the counter, thereby ignoring startup and wind-

down effects.

The results show that when concurrency is low, the spin-lock-based counter gives the highest throughput due to the simplicity of the spin lock algorithm. Nevertheless, when concurrency increases, throughput drops off dramatically, even for locking with exponential backoff. The MCS lock counter, the queue lock counter, and the message-based counter maintain essentially constant throughput as concurrency increases. This scalability can be attributed to queuing. In both the MCS-lock-based counter and the queue-based counter, queuing is explicitly performed in software. In the message-based counter, queuing occurs automatically in the processors' input message queues.

Because the queue-based counter combines transfer of the counter with transfer of the lock, it produces less network traffic, and outperforms the original MCS-lock counter by a factor of more than 2.5.

Finally, we observe that throughput increases with concurrency only for combining trees and for counting networks. This increase can be attributed to two factors: both techniques reduce contention, and both techniques permit parallel increments.

Optimizing combining trees and counting networks. We implement the combining tree and counting network counters using both shared-memory operations and message-passing. Figure 12 contrasts their performance, showing that the message-passing implementations have roughly twice the throughput.

There are two reasons for this performance difference. First, the message-passing implementation requires less communication because each balancer is always local to the processor that accesses it, and because traversing a data structure with messages is more efficient. Second, in the message-passing implementation, message receipt causes an interrupt whose handler is itself UN-interruptible by other messages, and therefore the interrupt handler does not require locks to ensure atomicity.

Saturation of counting networks. Figure 11 shows that the throughput of the 16-wide counting network dips at 64 processors. To determine whether this dip indicates that the counting network is saturating, we extended the simulation to 80 processors and tested counting networks with widths of 4, 8 and 16. Figure 13 shows that the 16-wide counting network does not saturate at 64 processors. We think the dip at 64 processors occurs because

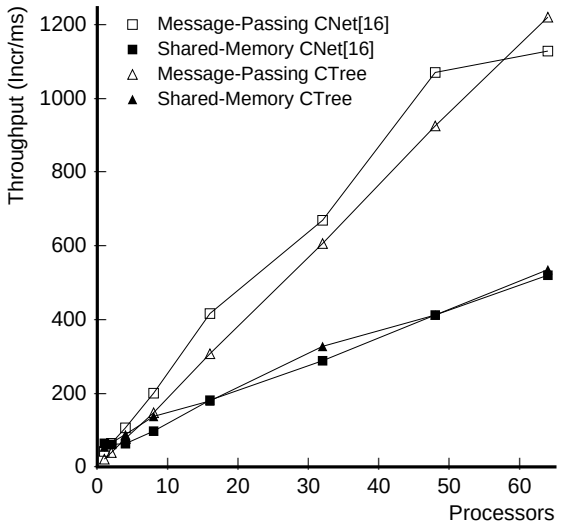

Figure 12: Comparing the throughput of combining trees (CTree) and counting networks (CNet) implemented with shared-memory and message-passing operations.

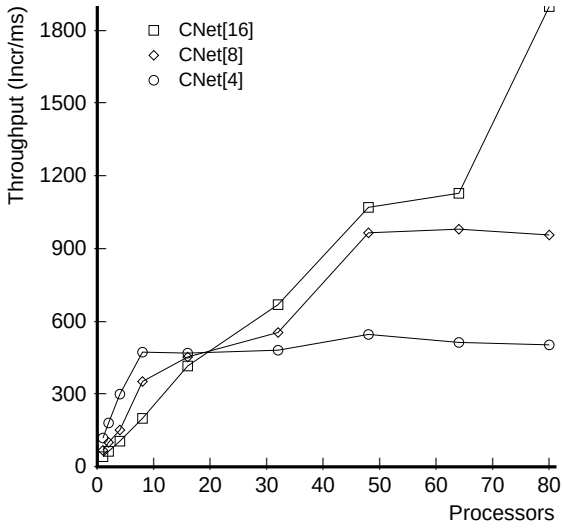

Figure 13: Throughput of various sizes of counting networks.

the 16-wide counting network contains 80 nodes, requiring more than one network node to be mapped onto a processor on a 64-processor machine. Figure 13 also shows the concurrency levels at which the smaller counting networks saturate.

4.2 Index Distribution Benchmark

We now look at the throughput of the shared counters when applied to index distribution. Compared to the counting benchmark, this benchmark provides a more varied load on the counters since each thread performs some computation in between increment requests. The amount of computation is varied by the parameter w: a higher w results in more computation. The effect of increasing w is to reduce concurrency (and contention) at the counter.

Figure 14 presents the results for a spin-lock-based counter, a message-based counter, a combining tree, and a counting network for various values of w. The elapsed times are plotted in a log-log graph so that linear speedups will show up as a straight line. Since the queue-based and MCS-lock-based

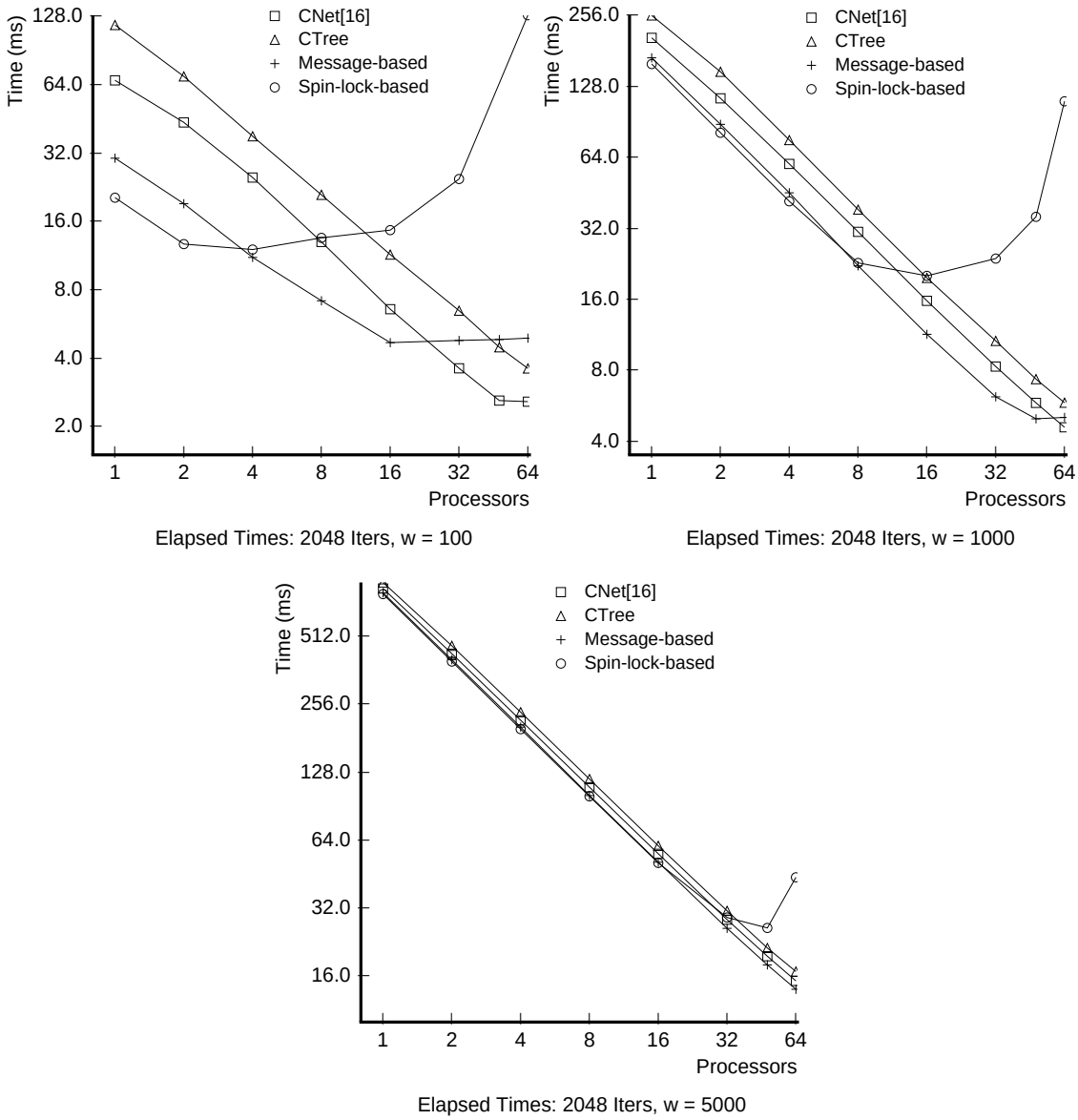

Figure 14: Elapsed time measurements of the index distribution benchmark.

counters have the same scaling behavior as the message-based counter, we omit them here.

For the spin-lock-based counter, performance degrades beyond a small number of processors. This degradation is worst when w is small. For the message-based counter, performance peaks and then degrades slightly beyond 16 processors when $w = 100$ and 48 processors when $w = 1000$. In contrast, both the combining tree and counting network sustain speedups on the benchmark all the way up to 64 processors.

Performance degrades drastically with the spin-lock-based counter because of contention, as can be expected from the throughput results presented earlier. While queuing reduces contention and prevents a major degradation of performance, sequential access to the message-based counter limits speedup when w, and thus computation grain size, is small. The only way to sustain speedups as more processors are added is to allow counting to occur in parallel, as in the combining tree and counting network.

4.3 Job Queue Benchmark

We now look at the performance of the shared counters when applied to a parallel job queue. Like the index distribution benchmark, this benchmark provides a varied load on the counters since each thread performs some computation in between accesses to the job queue. However, there are now two counters, one for enqueuing and one for dequeuing, and the operation includes an access to a shared data structure representing the job queue. Thus, this benchmark places less contention on the counters compared to the index distribution benchmark.

Figure 15 presents the results for a spin-lock-based counter, a message-based counter, a combining tree, and a counting network for various values of w. As before, the elapsed times are plotted in a log-log graph. Again, performance degrades drastically with the spin-lock-based counter and is limited with the message-based counter, reaffirming the observation it is necessary both to avoid contention and to permit parallelism to sustain speedups as more processors are added.

4.4 Combining Rates

When we compare the performance of the combining tree and the counting network in the index distribution and job queue benchmarks, we find that the counting network performs much better than can be expected from the

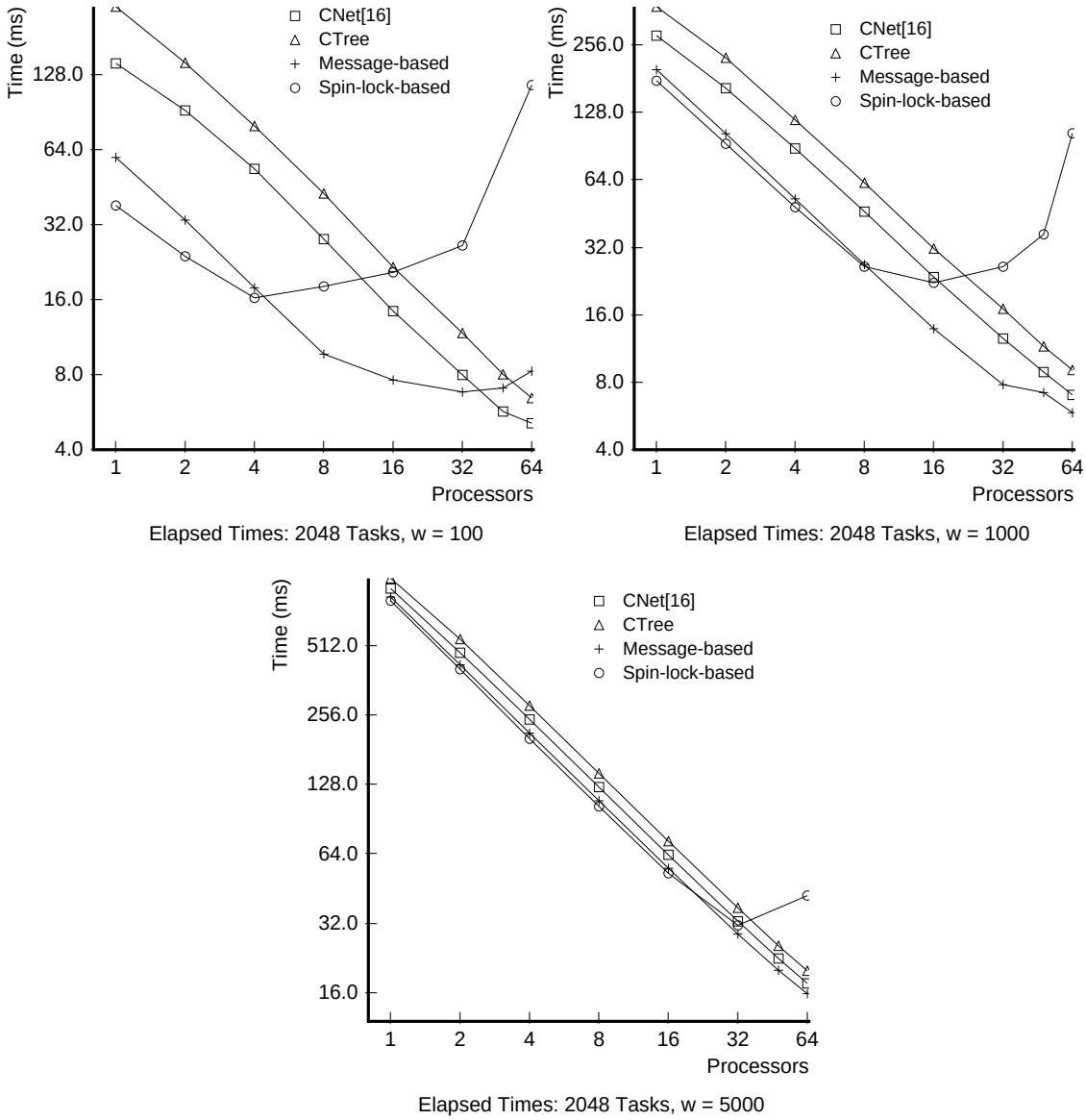

Figure 15: Elapsed time measurements of the job queue benchmark.

Concurrency	$w = 100$	$w = 1000$	$w = 5000$
1	0.0	0.0	0.0
2	10.3	2.0	0.3
4	26.1	8.5	2.2
8	40.3	19.9	4.8
16	50.4	31.7	10.6
32	55.3	39.1	18.5
48	54.2	40.0	15.6
64	56.5	39.8	18.7

Table 2: Combining rate (as a percentage) at combining tree nodes in the index distribution benchmark

throughput measurements in Figure 11. To investigate this phenomenon, we instrumented the simulation to monitor combining at the nodes of the combining tree. For the counting benchmark, we measured combining rates of close to 100% for 64 processors. Tables 2 and 3 summarize the results by presenting the percentage of arrivals at combining tree nodes that combine with some other arrival in the index distribution and job queue benchmarks.

From the data, we can see that as the rate of arrivals of increment requests is reduced, so does the rate of combining. In the combining tree algorithm, when a node misses a chance for combining, a request arriving later at that node must wait for the earlier request to ascend the tree and return before it can progress. We speculate that this sensitivity of combining trees to the arrival rate of increment requests degrades performance relative to counting networks.

4.5 Importance of Parallelism

Recall that counting networks and combining trees scale for two reasons: (1) distributing memory accesses reduces contention, and (2) parallelism increases throughput. To illustrate the relative importance of these two properties, we now investigate a counter implementation that has low contention, but does not attain a high degree of parallelism.

A counter is *linearizable* [23] if the values it returns are consistent with the real-time order of the matching requests. For example, linearizability ensures that if process p takes a value before process q requests a value,

Concurrency	$w = 100$	$w = 1000$	$w = 5000$
1	0.0	0.0	0.0
2	0.3	1.4	0.3
4	7.2	5.5	1.7
8	21.1	12.8	4.6
16	33.2	23.0	9.2
32	40.9	30.5	16.3
48	37.4	27.7	14.2
64	39.8	30.6	16.6

Table 3: Combining rate at combining tree nodes in the job queue benchmark

then p's value will be less than q's. The bitonic counting network is not linearizable, but it can be made linearizable by adding a simple *linearizing filter* to the network's output wires. The idea is simple: any token leaving the network waits until the token taking the next lower value exits. Although the solution introduces a sequential waiting chain, each processor will wait on a separate location, thus avoiding memory contention. (The linearizing filter can also be used to implement a general Fetch-and-Φ operation as in the combining tree.)

We construct the linearizable counting network for P processors from two component structures. One is the Bitonic counting network described above, and the other is a *linearizing filter* of width P. A linearizing filter is a P-element array of boolean values, called phase bits that are initially 0. Define the function $phase(v)$ to be $\lfloor (v/P) \rfloor \bmod 2$. We construct the linearizable network by having tokens first traverse the counting network and then access the waiting filter. When a token exits the counting network with value v, it awaits its predecessor by waiting until location $(v-1) \bmod P$ in the filter is set to $phase(v-1)$. When this event occurs, it notifies its successor by setting location v to $phase(v)$. It then returns its value.

Figure 16 demonstrates the importance of having both low contention and parallelization. It clearly shows that the throughput of the linearized counting network saturates beyond 16 processors even though contention in the linearized network is avoided. This emphasizes the importance of avoiding serialization in the design of shared data structures.

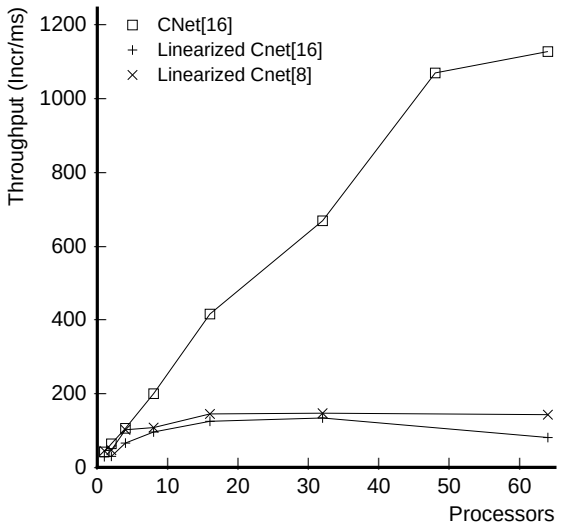

Figure 16: Throughput of a linearized counting network.

5 Conclusions

We have analyzed the throughput of five distinct counting techniques, each based on a technique proposed in the literature. We found that scalability for concurrent counting requires two logically distinct properties: avoidance of memory or interconnect contention, and allowing true concurrency among increment operations. The observed behaviors fell into three categories: (1) techniques whose throughput degraded as concurrency increased, (2) techniques whose throughput did not degrade, but leveled out starting at a low level of concurrency, and (2) techniques where throughput continued to increase with concurrency. The first category encompasses the lock-based counters, which suffer from contention as concurrency increases. The second category encompasses the message-based and queue-based counters, which do not suffer from contention, but do not allow concurrent access. The last category encompasses software combining trees and counting networks, which are the only techniques we observed to be truly scalable, since they avoid contention, and they permit concurrent access. Software combining trees were observed to be more sensitive to fluctuations in the arrival rates of requests. Both software combining trees and counting networks are significantly more efficient when implemented using message-passing instead of shared memory.

Our results suggest that distributed data structures designed to alleviate contention and enhance parallelism are the most promising approach to scalable synchronization. It would be interesting to see similar experiments for other problems, other benchmarks, and other architectures.

Acknowledgments

Our thanks to the members of the Alewife research group for ASIM and for putting up with the time-consuming simulations on the group's workstations. The Alewife project is funded by NSF grant # MIP-9012773 and DARPA contract # N00014-87-K-0825.

References

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th international symposium on computer architecture*, June 1989.

[2] A. Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[4] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks and multiprocessor coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, May 1991.

[6] K.E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference*, pages 334–338, 1968.

[7] B. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1991.

[8] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[9] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge MA, 1990.

[10] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, May 1992.

[11] E. Freudenthal and A. Gottlieb. Processor coordination with fetch-and-increment. In *Proceedings of the 4th ASPLOS*, April 91, pages 260–268.

[12] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd ASPLOS*, pages 64–75. ACM, April 1989.

[13] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[14] A. Gottlieb, 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*, 5(2):164–189, April 1983.

[15] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[16] M.P. Herlihy, B-H. Lim, and N. Shavit Low Contention Load Balancing on Large-Scale Multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1992, San Diego, CA.

[17] John Kubiatowicz, David Chaiken, and Anant Agarwal. The Alewife CMMU: Addressing the Multiprocessor Communications Gap. In *HOTCHIPS*, August 1994. To appear.

[18] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *International Supercomputing Conference (ICS) 1993*, Tokyo, Japan, July 1993. IEEE.

[19] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[20] J.M. Mellor-Crummey and T.J. LeBlanc. A software instruction counter. In *Proceedings of the 3rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 89.

[21] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[22] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System, Society of Photooptical Instrumentation Engineers, 1981, Vol 298, pages 241-248.

[23] M.P. Herlihy, N. Shavit, and O. Waarts. Linearizable Counting Networks In *Proceedings of the* 32^{nd} *Annual Symposium on Foundations of Computer Science,* San Juan, Puerto Rico, October 1991, pp. 526-535.

[24] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[25] P.C Yew, N.F. Tzeng, and D.H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.