INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reducing the Overhead of Sharing on Shared Memory Multiprocessors

by

Maged M. Michael

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Michael L. Scott

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1997

To my parents, Hilda and Milad Michael

Curriculum Vitae

Maged Michael was born in Alexandria, Egypt. In 1988, he received a Bachelor of Science degree with honors from Alexandria University in Computer Science. In 1992, he entered the graduate program at the Department of Computer Science, University of Rochester and received a Master of Science degree in Computer Science in 1994. His research focused on multiprocessor synchronization techniques and algorithms, cache coherence protocols, and memory hierarchy architecture. At Rochester he worked closely with his Ph.D. advisor Professor Michael L. Scott on the above issues. In 1995 he interned at the Scalable Systems Division of Intel Corporation, and in 1996 he interned at the IBM Thomas J. Watson Research Center.

Acknowledgments

I owe almost everything I know about parallel computing, and about doing research in computer science to my mentor and advisor, Michael Scott. I thank him for all the guidance, support, flexibility, encouragement, and for caring about my career goals.

The other members of my committee, Alexander Albicki, Tom Leblanc, and Wei Li, also deserve many thanks for the helpful discussions and suggestions throughout the course of my Ph.D. work. Tom contributed directly to my development as a researcher by running the boot camp called CSC 400. My thanks to the department's staff for being the most competent and friendly I have ever seen.

In addition to Michael Scott, my thanks to all the other collaborators in the systems group especially Galen Hunt and Srinivasan Parthasarathy, and to Jack Veenstra for creating MINT. I also had the pleasure of collaborating with Anthony-Trung Nguyen, Arun Sharma, and Josep Terrellas from the University of Illinois at Urbana Champaign, and John Carbajal, Dave Archer, and many others at Intel

Corporation.

From the IBM Thomas J. Watson Research Center I would like to thank for a wonderful research environment, Michael Rosenfield, Ashwini Nanda, Beng-Hong Lim, Moriyoshi Ohara, Pradip Bose, Kattamuri Ekanadham, and all the other members of the parallel systems departments.

Special thanks to Shumin for all the support and encouragement, and to my friends Olac, Justin, Ramesh, Raj, Choh Man, Martin, Michal, Eric, Marius, and all the friends who made my stay in Rochester a pleasant experience.

My deepest gratitude to my parents for their unconditional love and support throughout my life, and for showing me firsthand what it is like to be dedicated and distinguished scientists. I owe my pursuit of a career in research to their inspiring example.

This work was supported in part by NSF Institutional Infrastructure grants nos. CDA–88–22724, CCR–93–19445, and CDA–94–01142, by ONR research grant no. N00014–92–J–1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930), and by IBM Corporation.

Abstract

Shared memory provides an intuitive and flexible programming model for parallel application developers relative to the message passing programming model. However, the need to keep shared data structures consistent via synchronization mechanisms and cache coherent shared memory, entails substantial overheads. The goal of this work is to characterize and improve the performance of shared memory multiprocessors by reducing the overheads associated with synchronization and cache coherence on shared memory multiprocessor systems. To that end, we present innovative techniques and quantitative experimental performance evaluation and analysis.

The contributions of this work include new fast shared data structure algorithms, including a link-based shared queue that outperforms all known alternatives. We also study the performance of alternative synchronization techniques on multiprogrammed systems. The study demonstrates the superior performance of data-structure-specific non-blocking algorithms over lock-based techniques on multiprogrammed as well as dedicated systems, and the importance of universal atomic primitives such as `compare_and_swap` and the pair `load_linked` and

`store_conditional`. We propose cache coherence protocols for the hardware implementation of these primitives on distributed shared memory multiprocessor systems and evaluate their performance, recommending an implementation of `compare_and_swap` for future architectures. Finally, we study the architecture of coherence controllers which is a crucial factor in the performance of cache coherence mechanisms and overall shared memory system performance. Our study demonstrates that the bandwidth of coherence controllers is the main bottleneck for SMP-based CC-NUMA multiprocessor systems. The study also demonstrates the significant performance benefits of using multiple protocol engines and introduces a new measure of parallel application demand for coherence controller bandwidth. This metric can be used to quickly predict the performance of large parallel applications.

Table of Contents

List of Figures

List of Tables

1 Introduction

Parallel application developers use two main programming models to express parallel algorithms: message passing and shared memory. In message-passing systems, the programmer's view is of a set of separate computers that communicate only by exchanging explicit messages. Message-passing, in general, requires the programmer (or the compiler) to control communication explicitly between the parallel application processes. In shared memory systems, memory is accessible to all processors, and processors communicate through shared variables.

Message passing allows the programmer to optimize the application for minimal communication, and accordingly achieve high speedup on multiprocessor systems. The main problem with message passing is that the programmer has to use different models for intra- and inter-processor computing, which is unintuitive. In addition, it is sometimes impossible for the programmer to anticipate the optimal communication patterns between the parallel processes for applications with

input-dependent control structure. Also, it is very difficult to employ fine-grain parallelism and dynamic load balancing under message passing.

Shared memory, on the other hand, provides an intuitive and easy programming model for parallel application developers, as it frees the programmer from the burden of planing explicit communication between the application processes. Shared memory allows each application process to consider the whole shared memory as part of its address space. However, since multiple processes are allowed to access and possibly modify the shared data concurrently, explicit synchronization is needed to maintain the consistency of shared data structures. Explicit synchronization can be a significant source of overhead on shared memory systems relative to message passing systems.

Another significant source of overhead on shared memory multiprocessors is cache coherence. Due to the large disparity between the access times to processor caches and main memory, processor caches are essential in modern multiprocessor and uniprocessor systems. Under shared memory, multiple processors can access and replicate a copy of the same memory location in their caches. Replication introduces the cache coherence problem: the need to keep copies consistent in the face of local changes.

The goal of this work is to characterize and improve the performance of shared memory multiprocessors by reducing the overheads associated with synchronization and cache coherence. We achieve this goal through innovative techniques, and

quantitative experimental performance evaluation and analysis. We use native execution on a Silicon Graphics Challenge multiprocessor to study the performance of software-based techniques, and execution-driven simulation to study the performance of architectural designs.

After discussing the overheads of synchronization in Section 1.1 and cache coherence in Section 1.2, we present an overview of the main contributions of this work in Section 1.3, and the outline of the rest of the dissertation in Section 1.4.

1.1 Synchronization

On shared memory multiprocessors, processes communicate through shared data structures. Synchronization techniques are needed to maintain the consistency of these data structures under arbitrary concurrent updates. The overhead of naive approaches to synchronization is one of the main sources of overhead on shared memory multiprocessor systems. We study the performance of important common concurrent data structures under varying levels of contention and processor scheduling policies, and propose new implementations.

The most common technique for synchronizing concurrent access to shared data structures is to enclose each operation on the data structure in a critical section protected by a simple mutual exclusion lock, commonly implemented using the `test_and_set` atomic primitive. Such locks are not scalable, they limit concur-

rency, and they cause the application to suffer significant performance degradation under multiprogramming.

Researchers have proposed several techniques to address these problems. Queue-based locks [5; 21; 52; 56] were proposed mainly to address the scalability of mutual exclusion locks. Reactive synchronization [49] was proposed to achieve near optimal performance under various levels of contention. Reader-writer locks [57] and multiple-lock data structures [43; 47; 76] were proposed to increase concurrency and hence reduce serialization. Preemption-safe locks [7; 15; 41; 53; 67] and non-blocking synchronization [3; 8; 9; 29; 45; 54; 77] were proposed to protect the performance of shared data structures from the adverse effects of multiprogramming.

In this dissertation we address the issues of concurrency and robustness under multiprogramming for important data structures. We present and study the performance of new multiple-lock algorithms for priority queue heaps and link-based queues, and a non-blocking algorithm for link-based queues. Also, we study the relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed systems with queues, stacks, counters, and priority queues.

1.2 Cache Coherence

Cache coherence frees the programmer from the burden of ensuring the consistency of shared data through explicit message passing. On bus-based symmetric multiprocessors (SMPs), cache coherence is achieved through broadcast-based snooping protocols. However, the centralized nature of a shared bus limits the scalability of bus-based systems. Cache coherence protocols on distributed shared memory (DSM) systems are more complex and, typically, they involve exchanging many more messages that on a comparable message passing system. Cache coherence is one of the main sources of overhead on DSM multiprocessors.

The main issues in cache coherence performance on DSM systems are protocol design, flexibility in protocol design, and the cost-effectiveness of hardware support for cache coherence. Researchers have proposed several classes of software- and hardware-based cache coherence protocols. Flexibility in protocol design varies from all-software cache coherence, to programmable hardware protocol processors, to custom hardware finite state machines. We concentrate, in this dissertation, on hardware cache coherent DSM systems. We present and evaluate the impact of various cache coherence policies on the implementation and performance of atomic primitives, and we study the performance implications of alternative coherence controller architectural designs.

1.3 Contributions

The contributions of this dissertation include:

- Presenting new algorithms for:
  - Multiple-lock priority queue heaps [33]. Our algorithm allows $O(M)$ concurrency, where M is the size of the heap. The algorithm outperforms other multiple-lock algorithms in all cases, and outperforms single lock priority queue heaps in the case of high contention with large heap sizes.
  - Non-blocking link-based queues [60]. Our algorithm is simple, fast, and practical. It is the best known link-based implementation on dedicated as well as multiprogrammed systems, under both high and low contention.
  - Two-lock link-based queues [60]. Our algorithm allows complete concurrency between enqueue and dequeue operations on non-empty queues. It outperforms single-lock link-based queue implementations under high contention.
- Demonstrating the superior performance of special-purpose non-blocking implementations of common data structures such as queues, stacks, and counters, over lock-based implementations on multiprogrammed as well as

dedicated systems [62]. Our study also demonstrates the superior performance of preemption-safe locking with respect to ordinary (preemption-oblivious) locking on multiprogrammed systems.

- Presenting and evaluating implementations of the atomic primitives `fetch_and_Φ`, `compare_and_swap`, and `load_linked/store_conditional` in the context of directory-based cache coherence protocols on DSM multiprocessors [59]. In particular, our study recommends implementing `compare_and_swap` with a write-invalidate protocol in combination with a load exclusive primitive.

- Evaluating the performance of coherence controller architectures [61] on SMP-based cache-coherent non-uniform memory access (CC-NUMA) systems, demonstrating the following results:

  - The occupancy (bandwidth) of protocol engines is a major bottleneck on CC-NUMA systems.

  - Using commodity protocol processors in the coherence controller can lead to significant performance degradation relative to using custom hardware controllers.

  - Using multiple protocol engines in the coherence controller increases bandwidth and improves the performance of the system.

  - The relative performance of systems with various coherence controller

designs can be predicted using a new measure of application demand for coherence controller bandwidth namely, RCCPI.

1.4 Outline

The rest of the dissertation is organized as follows. Chapter 2 presents our multiple-lock and non-blocking priority queue heaps and link-based queues. Chapter 3 describes our study of the performance of non-blocking synchronization and preemption-safe locking on multiprogrammed systems. Chapter 4 considers the implementation of atomic primitives in the context of hardware cache coherence protocols on distributed shared memory multiprocessors. Our study of the performance of coherence controller architectures appears in Chapter 5. We summarize our results and recommendations in Chapter 6.

2 Shared Data Structures

Parallel applications with significant amounts of concurrent access to naive implementations of shared data structures can suffer substantial performance losses due to synchronization. To address this problem, we present three novel algorithms: one for shared priority queue heaps and two for link-based queues.

The priority queue heap algorithm uses multiple locks to allow more concurrent access, thus increasing the throughput of the data structure.

The shared queue algorithms achieve performance superior to that of other queue implementations as a result of their simplicity. One algorithm uses two locks to allow concurrency between enqueues and dequeues. The other algorithm is non-blocking and accordingly is immune to the adverse effects of multiprogramming on the performance of shared data structures, and allows enqueues and dequeues to proceed concurrently. The latter algorithm outperforms all other known implementations of a link-based queue.

2.1 Shared Priority Queue Heap Algorithm

2.1.1 Introduction

The heap data structure is widely used as a priority queue [14]. The basic operations on a priority queue are *insert* and *delete*. *Insert* inserts a new item in the queue and *delete* removes and returns the highest priority item from the queue. A heap is a binary tree with the property that the key at any node has higher priority than the keys at its children (if they exist). An array representation of a heap is the most space efficient: the root of the heap occupies location 1 and the left and right children of the node at location i occupy locations $2i$ and $2i + 1$, respectively. No items exist in level l of the tree unless level $l - 1$ is completely full.

Many applications (e.g. heuristic search algorithms, graph search, and discrete event simulation [63; 71]) on shared memory multiprocessors use shared priority queues to schedule sub-tasks. In these applications, items can be inserted and deleted from the heap by any of the participating processes. The simplest way to ensure the consistency of the heap is to serialize the updates by putting them in critical sections protected by a mutual exclusion lock. This approach limits concurrent operations on the heap to one. Since updates to the heap typically modify only a small fraction of the nodes, more concurrency should be achievable by allowing processes to access the heap concurrently as long as they do not

interact with each other.

Biswas and Browne [10] proposed a scheme that allows many insertions and deletions to proceed concurrently. Their scheme relies on the presence of maintenance processes that dequeue sub-operations from a FIFO work queue. Sub-operations are placed on the work queue by the processes performing insert and delete operations. The work queue is used to avoid deadlock due to insertions and deletions proceeding in opposite directions in the tree. The need for a work queue and maintenance processes causes this scheme to incur substantial overhead. Rao and Kumar [72] present another scheme that avoids deadlock by using top-down insertions, where an inserted item has to traverse a path through the whole height of the heap (insertions in a traditional sequential heap proceed bottom-up). Jones [37] presents a concurrent priority queue algorithm using skew heaps.[1] He notes that top-down insertions in array-based heaps are inefficient, while bottom-up insertions would cause deadlock if they collide with top-down deletions without using extra server processes.

This section presents a new concurrent priority queue heap algorithm that addresses the problems encountered in previous research. On large heaps the algorithm achieves significant performance improvements over both the serialized single-lock algorithm and the algorithm of Rao and Kumar, for various inser-

[1] An array based heap can be considered as a binary tree that is filled at all levels except possibly the last level. In skew heaps this restriction is relaxed; the representative binary tree need not be filled at all the intermediate levels.

tion/deletion workloads. For small heaps it still performs well, but not as well as the single-lock algorithm. The new algorithm allows concurrent insertions and deletions in opposite directions, without risking deadlock and without the need for special server processes. It also uses a "bit-reversal" technique to scatter accesses across the fringe of the tree to reduce contention.

2.1.2 The Algorithm

The new algorithm augments the standard heap data structure [14] with a mutual-exclusion lock on the heap's size and locks on each node in the heap. Each node also has a tag that indicates whether it is empty, valid, or in a transient state due to an update to the heap by an inserting process. Nodes that contain no data are tagged **EMPTY**. Nodes that are available for deletion are tagged **AVAILABLE**. A node that has been inserted, and is being moved into place, is tagged with the process identifier (*pid*) of the inserting process.

A delete operation in the new algorithm, as in the sequential algorithm, starts by reading the data and priority of the root of the heap and then replacing them with those of as rightmost node in the lowest level of the heap. Then, the delete operation "heapifies" the heap. It compares the priority of the root with that of each of its children (if any). If necessary, it swaps the root item with one of its children in order to ensure that none of the children has priority higher than the root. If no swapping is necessary the delete operation is complete; it returns

```
record data_item   {lock := FREE; tag := EMPTY; priority := 0}
record heap        {lock := FREE; bit_reversed_counter size; data_item items[]}
define LOCK(x) as lock(heap.items[x].lock)
define UNLOCK(x) as unlock(heap.items[x].lock)
define TAG(x) as heap.items[x].tag
define PRIORITY(x) as heap.items[x].priority

procedure concurrent_insert(priority, heap)
  # Insert new item at bottom of the heap.
  lock(heap.lock); i := bit_reversed_increment(heap.size); LOCK(i); unlock(heap.lock)
  PRIORITY(i) := priority; TAG(i) := pid; UNLOCK(i)

  # Move item towards top of heap while it has a higher priority than its parent.
  while i > 1 do
    parent := i / 2; LOCK(parent); LOCK(i)
    if TAG(parent) = AVAILABLE and TAG(i) = pid then
      if PRIORITY(i) > PRIORITY(parent) then
        swap_items(i, parent); i := parent
      else
        TAG(i) := AVAILABLE; i := 0
    else if TAG(parent) = EMPTY then
      i := 0
    else if TAG(i) ≠ pid then
      i := parent
    UNLOCK(i); UNLOCK(parent)
  enddo
  if i = 1 then
    LOCK(i)
    if TAG(i) = pid then
      TAG(i) := AVAILABLE
    UNLOCK(i)
```

Figure 2.1: Concurrent insert operation. For conciseness, we treat *priority* as if it were the only datum in each data_item.

the data that was originally in the root. Otherwise, the operation recursively "heapifies" the subheap rooted at the swapped child. To handle concurrency all these steps are performed under the protection of the locks on the individual nodes and a lock on the size of the heap. In each step of the heapify operation, the lock

```
function concurrent_delete(heap)
  # Grab an item from the bottom of the heap to replace the to-be-deleted top item.
  lock(heap.lock);
    bottom := bit_reversed_decrement(heap.size); LOCK(bottom);
  unlock(heap.lock)
  priority := PRIORITY(bottom); TAG(bottom) := EMPTY; UNLOCK(bottom)

  # Lock first item. Stop if it was the only item in the heap.
  LOCK(1); if TAG(1) = EMPTY then UNLOCK(1); return priority

  # Replace the top item with the item stored from the bottom.
  swap(priority, PRIORITY(1)); TAG(1) := AVAILABLE

  # Adjust the heap starting at the top.
  # We always hold a lock on the item being adjusted.
  i := 1
  while (i < MAX_SIZE / 2) do
    left := i * 2; right := i * 2 + 1; LOCK(left); LOCK(right)
    if TAG(left) = EMPTY then
      UNLOCK(right); UNLOCK(left); break
    else if TAG(right) = EMPTY or PRIORITY(left) > PRIORITY(right) then
      UNLOCK(right); child := left
    else
      UNLOCK(left); child := right

    # If the child has a higher priority than the parent then swap them. If not, stop.
    if PRIORITY(child) > PRIORITY(i) then
      swap_items(child, i); UNLOCK(i); i := child
    else
      UNLOCK(child); break
  enddo
  UNLOCK(i)
  return priority
```

Figure 2.2: Concurrent delete operation.

of the subtree root is already held. It is not released until the end of that step. Prior to comparing priorities, the locks of the children are acquired. If swapping is performed, the lock on the swapped child is retained through the next recursive

heapify step, and the locks on the root and the unswapped child are released. Otherwise, all the locks are released, and the delete operation completes.

An insert operation starts by inserting the new data and priority in the lowest level of the heap. If the inserted node is the root of the heap, then the insert operation is complete. Otherwise, the operation compares the priority of the inserted node to that of its parent. If the child's priority is higher than that of its parent, then the two items are swapped, otherwise the insert operation is complete. If swapping was necessary, then the same steps are applied repeatedly bottom-up until reaching a step in which no swapping is necessary, or the inserted node has become the root of the heap. To handle concurrency, all these steps are performed under the protection of the locks and tags on the individual nodes and the lock on the size of the heap. In every step of the bottom-up comparison, the lock of the parent is acquired first, followed by the lock on the inserted node. After comparison and swapping (if necessary), both locks are released. Locks are acquired in the same order as in the delete operation, parent-child, to avoid deadlock. This mechanism requires releasing and then acquiring the lock on the inserted item between successive steps, which opens a window of vulnerability during which the inserted item might be swapped by other concurrent operations. Tags are used to resolve these situations.

An insert operation tags the inserted item with its *pid*. In every step, an insert operation can identify the item it is moving up the heap even if the item has been

swapped upwards by a deletion. In particular, tags are used in the following manner:

- If the tag of the parent node is equal to **AVAILABLE** and the tag of the current node is equal to the insert operation's *pid*, then no interference has occurred and the insertion step can proceed normally.
- If the tag of the parent node is equal to **EMPTY**, then the inserted item must have been moved by a delete operation to the root of the heap. The insert operation is complete.
- If the tag of the current node is not equal to the operation's *pid*, then the inserted item must have been moved upwards by a delete operation. The insert operation moves upward in pursuit of the inserted item.

In some definitions of heaps [14], all nodes in the last level of the heap to the left of the last item have to be non-empty. Since this is not required by priority queue semantics, in the new algorithm we chose to relax this restriction to reduce lock contention, and thereby permit more concurrency. Under our relaxed model, consecutive insertions traverse different sub-trees by using a "bit-reversal" technique similar to that of an FFT computation [14]. For example, in the third level of a heap (nodes 8-15, where node 1 is the root), eight consecutive insertions would start from the nodes 8, 12, 10, 14, 9, 13, 11, and 15, respectively. Notice that for any two consecutive insertions, the two paths from each of the bottom level

nodes to the root of the heap have no common nodes other than the root. This lack of overlap serves to reduce contention for node locks. Consecutive deletions from the heap follow the same pattern, but in reverse order. The relation between the indices of parents and children remains as it is in heaps without bit reversal. The children of node i are nodes $2i$ and $2i+1$, and the parent of node $i > 1$ is node $i/2$. Moreover, if a node has only one child, it is still $2i$, never $2i+1$.

Since insertions in the new algorithm do not have to traverse the whole height of the heap, they have a lower bound of $\Omega(1)$ time, while the algorithm due to Rao and Kumar requires $\Omega(\log M)$ time for insertions (top-down) in a heap of size M, as insertions have to traverse the entire height of the heap. In addition to reducing traversal overhead, the bottom-up insertion approach of the new algorithm reduces contention on topmost nodes.

We next consider the space requirements for algorithms under consideration. Let M be the maximum number of nodes in the heap, and P the maximum number of processes operating on the heap. Assume that each lock requires one bit of memory. The new algorithm requires 1 bit for the lock on the heap size variable, $3 \log M$ bit-reversal bits, and $1 + \log P$ lock and tag bits per node, for a total of $1 + 3 \log M + (1 + \log P)M$ bits of memory. The single lock algorithm requires 1 bit of memory for the single lock. Rao and Kumar's algorithm requires 3 bits per node for a total of $3M$ bits of memory. If bit reversal were added to Rao and Kumar's algorithm, it would require $3 \log M$ extra bits, for a total of

```
record bit_reversed_counter
  {counter := 0;  reversed := 0;  high_bit := -1}

function bit_reversed_increment(c)
  c.counter := c.counter + 1
  for bit := c.high_bit - 1 to 0 step -1
    c.reversed := not(c.reversed, bit)
    if test(c.reversed, bit) = TRUE then
      break
  if bit < 0 then
    c.reversed := c.counter;  c.high_bit := c.high_bit + 1
  return c.reversed

function bit_reversed_decrement(c)
  c.counter := c.counter - 1
  for bit := c.high_bit - 1 to 0 step -1
    c.reversed := not(c.reversed, bit)
    if test(c.reversed, bit) = FALSE then
      break
  if bit < 0 then
    c.reversed := c.counter;  c.high_bit := c.high_bit - 1
  return c.reversed
```

Figure 2.3: A bit-reverse counter.

$3 \log M + 3M$ bits of memory. The single lock algorithm is significantly more space efficient than the multiple lock algorithms. Rao and Kumar's algorithm requires less space than the new algorithm ($\Theta(M)$ for the former compared to $\Theta(M \log P)$ for the latter). In practice, however, bit packing results in false sharing in cache-coherent systems, and should therefore be avoided. If overhead bits for different nodes occupy different memory words, and if the number of processes operating on the heap does not exceed $2^n - 2$, where n is the number of bits per memory word, then the space overhead of the new algorithm is the same as that of Rao and Kumar's algorithm, except for three words for the bit-reverse counter.

Figures 2.1 and 2.2 present pseudo code for the insert and delete operations of the new algorithm, respectively. Initially, all locks are free, all node tags are set to `EMPTY`, and the number of elements in the heap is zero.

Bit reversals can easily be calculated in $O(n)$ time, where n is the number of bits to be reversed. For long sequences of increments only or decrements only, we can improve this bound to an amortized time of $O(1)$ by remembering the high-order bit (see Figure 2.3). Alternating increments and decrements may still require $O(n)$ time.

2.1.3 Experimental Methodology

We use a 12-processor Silicon Graphics Challenge multiprocessor to compare the performance of the new algorithm, the single-lock algorithm, and Rao and Kumar's algorithm. We tried the latter both with and without adding our bit-reversal technique, in order to determine if it suffices to improve performance. For mutual exclusion we used test-and-test-and-set locks with backoff using the MIPS R4000 `load-linked` and `store-conditional` instructions. On small-scale multiprocessors like the Challenge, these locks have low overhead compared to other more scalable locks. [56].

To evaluate the performance of the algorithms under different levels of contention, we varied the number of processes in our experiments. Each process runs on a dedicated processor in a tight loop that repeatedly updates a shared heap.

Thus, in our experiments the number of processors corresponds to the level of contention. We believe these results to be comparable to what would be achieved with a much larger number of processes, each of which was doing significant real work between queue operations. In all experiments, processors are equally loaded. We studied the performance under workloads of insertions only, deletions only, and various mixed insert/delete distributions. We also varied the initial number of full levels in the heap before starting time measurements to identify performance differences with different heap sizes. For the experiments we used workloads of around 100,000 to 200,000 heap operations. Experiments with smaller workloads are too fast to time. Inserted item priorities were chosen from a uniform random distribution on the domain of 32-bit integers.

The sources for all the algorithms were carefully hand-optimized. For example in the multiple-lock algorithms we changed the data layout to reduce the effect of false sharing. This optimization was not applied to the single lock algorithm as it does not support concurrent access; aligning data to cache lines would only increase the total number of cache misses. We believe we have implemented each algorithm as well as is reasonably possible, resulting in fair comparisons.

Figures 2.4a and 2.4b show the time taken to perform 100,000 insertions and deletions on a heap with 17 full levels. Figure 2.5 shows the time taken to perform 10,000 sets of 10 insertions and 10 deletions on an empty heap. Figures 2.6a and 2.6b show the time taken to perform 100,000 insert/delete pairs on a 7-level-

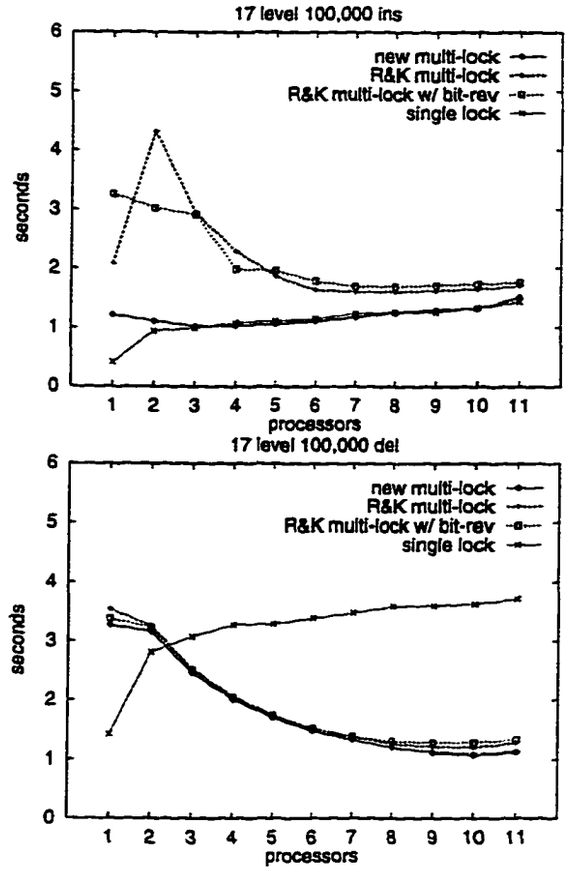

Figure 2.4: Performance results for a) 100,000 insertions and b) 100,000 deletions.

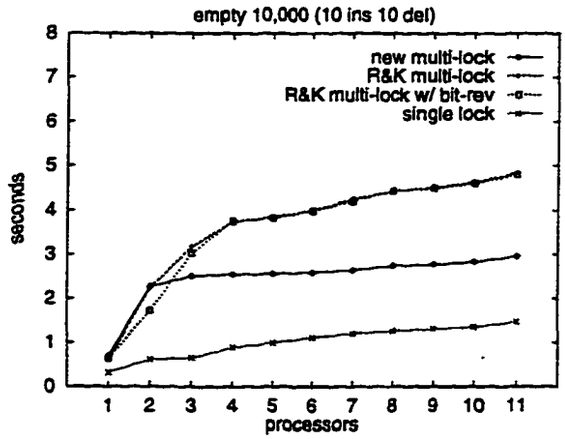

Figure 2.5: Performance results for 10,000 sets of 10 insertions and 10 deletions on an empty heap.

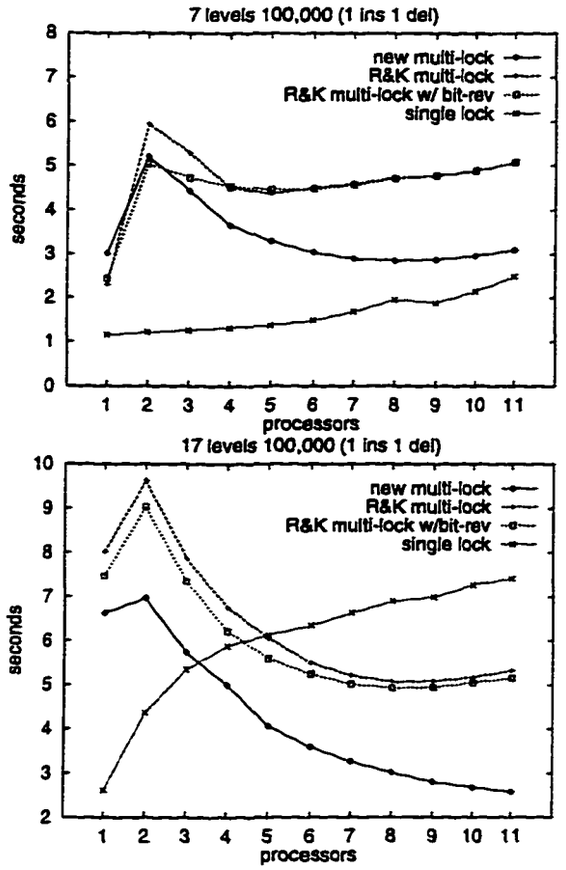

Figure 2.6: Performance results for a) 100,000 insert/delete pairs on a 7-level-full heap and b) 100,000 insert/delete pairs on a 17-level-full heap.

full heap and a 17-level-full heap.

In the case of insertions only (Figure 2.4a), the single-lock and the new algorithm have better performance because insertions do not have to traverse the whole height of the tree (as they do in Rao and Kumar's algorithm), and most inserted items settle in the two bottom-most levels of the heap. Insert operations for the single-lock algorithm in this case are fast enough that greater potential for concurrency in the new multi-lock algorithm does not help much.

In the case of deletions only (Figure 2.4b), the multi-lock algorithms outperform the single-lock algorithm. This is because most deletions have to traverse the whole height of the tree and may not traverse the same path each time. As a result, the concurrency permitted in the multi-lock algorithms is higher and outweighs the overhead of locking, since there is little contention along the paths. Deletions in the new algorithm proceed top-down, similar to deletions in Rao and Kumar's algorithm; therefore the two algorithms display similar performance.

In the case of alternating insertions and deletions on an initially empty heap (Figure 2.5), the height of the heap is very small. The single-lock algorithm outperforms the other algorithms because it has low overhead and there is little opportunity for the multi-lock algorithms to exploit concurrency. Comparing the new algorithm with that of Rao and Kumar, we find that the new algorithm yields better performance as it suffers less from contention on the topmost nodes of the heap. Note that after several insert/delete cycles, the items remaining in the heap tend to have low priorities, so new insertions have to traverse most of the path to the root in the new algorithm. This means that the performance advantage of the new algorithm over that of Rao and Kumar in this case is due more to reduced contention for the topmost nodes of the tree (due to opposite directions for insertion and deletion) than to shorter traversals.

In the case of alternating insertions and deletions on a 7-level-full heap (Figure 2.6a), the heap height remains almost constant. The single-lock algorithm

outperforms the others due to its low overhead, but the difference between it and the new algorithm narrows as the level of contention increases, since 7 levels provide the new algorithm with reasonable opportunities for concurrency. Rao and Kumar's algorithm suffers from high contention on the topmost nodes.

In the case of alternating insertions and deletions on a 17-level-full heap (Figure 2.6b), the larger heap height makes concurrency, rather than locking overhead, the dominant factor in performance. The multi-lock algorithms therefore perform better than the single-lock algorithm. As in the case of the empty and 7-level-full heaps, new insertions tend to have higher priorities than the items already in the heap, and tend to settle near the top of the heap. In spite of this, the new algorithm outperforms that of Rao and Kumar because of reduced contention on the topmost nodes.

2.1.4 Summary

We have presented a new algorithm that uses multiple mutual exclusion locks to allow consistent concurrent access to array-based priority queue heaps. The new algorithm avoids deadlock among concurrent accesses without forcing insertions to proceed top-down [72], or introducing a work queue and extra processes [10]. Bottom-up insertions reduce contention for the topmost nodes of the heap, and avoid the need for a full-height traversal in many cases. The new algorithm also uses bit-reversal to increase concurrency among consecutive insertions, allowing

them to follow mostly-disjoint paths. Empirical results, comparing the new algorithm, the single-lock algorithm, and Rao and Kumar's top-down insertion algorithm [72] on a SGI Challenge machine, show that the new algorithm provides reasonable performance on small heaps, and significantly superior performance on large heaps under high levels of contention.

2.2 Shared Queue Algorithms

2.2.1 Introduction

Concurrent FIFO queues are widely used in parallel applications and operating systems. To ensure correctness, concurrent access to shared queues has to be synchronized. Generally, algorithms for concurrent data structures, including FIFO queues, fall into two categories: *blocking* and *non-blocking*. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Possible sources of delay include processor scheduling preemption, page faults, and cache misses. Non-blocking

algorithms are more robust in the face of these events.

Many researchers have proposed lock-free algorithms for concurrent FIFO queues. Hwang and Briggs [34], Sites [79], and Stone [84] present lock-free algorithms based on `compare_and_swap`.[2] These algorithms are incompletely specified; they omit details such as the handling of empty or single-item queues, or concurrent enqueues and dequeues. Lamport [46] presents a wait-free algorithm that restricts concurrency to a single enqueuer and a single dequeuer.[3]

Gottlieb *et al.* [20] and Mellor-Crummey [55] present algorithms that are lock-free but not non-blocking: they do not use locking mechanisms, but they allow a slow process to delay faster processes indefinitely.

Treiber [86] presents an algorithm that is non-blocking but slow: a dequeue operation takes time proportional to the number of the elements in the queue. Herlihy [30]; Prakash, Lee, and Johnson [69]; Turek, Shasha, and Prakash [88]; and Barnes [9] propose general methodologies for generating non-blocking versions of sequential or concurrent lock-based algorithms. However, the resulting implementations are generally slow compared to specialized algorithms.

Massalin and Pu [54] present lock-free algorithms based on a `double_compare_and_swap` primitive that operates on two arbitrary memory locations simultane-

[2] `Compare_and_swap`, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred.

[3] A *wait-free* algorithm is both non-blocking and starvation free: it guarantees that every active process will make progress within a bounded number of time steps.

ously, and that seems to be available only on later members of the Motorola 68000 family of processors. Herlihy and Wing [25] present an array-based algorithm that requires infinite arrays. Valois [89] presents an array-based algorithm that requires either an unaligned `compare_and_swap` (not supported on any architecture) or a Motorola-like `double_compare_and_swap`.

Stone [82] presents a queue that is lock-free but non-linearizable[4] and not non-blocking. It is non-linearizable because a slow enqueuer may cause a faster process to enqueue an item and subsequently observe an empty queue, even though the enqueued item has never been dequeued. It is not non-blocking because a slow enqueue can delay dequeues by other processes indefinitely. Our experiments also revealed a race condition in which a certain interleaving of a slow dequeue with faster enqueues and dequeues by other process(es) can cause an enqueued item to be lost permanently. Stone also presents [83] a non-blocking queue based on a circular singly-linked list. The algorithm uses one anchor pointer to manage the queue instead of the usual head and tail. Our experiments revealed a race condition in which a slow dequeuer can cause an enqueued item to be lost permanently.

Prakash, Lee, and Johnson [68; 70] present a linearizable non-blocking algorithm that requires enqueuing and dequeuing processes to take a snapshot of the queue in order to determine its "state" prior to updating it. The algorithm

[4]An implementation of a data structure is linearizable if it can always give an external observer, observing only the abstract data structure operations, the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response [28].

achieves the non-blocking property by allowing faster processes to complete the operations of slower processes instead of waiting for them.

Valois [89; 90] presents a list-based non-blocking algorithm that avoids the contention caused by the snapshots of Prakash *et al.*'s algorithm and allows more concurrency by keeping a dummy node at the head (dequeue end) of a singly-linked list, thus simplifying the special cases associated with empty and single-item queues (a technique suggested by Sites [79]). Unfortunately, the algorithm allows the tail pointer to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or re-using dequeued nodes. If the tail pointer lags behind and a process frees a dequeued node, the linked list can be broken, so that subsequently enqueued items are lost. Since memory is a limited resource, prohibiting memory reuse is not an acceptable option. Valois therefore proposes a special mechanism to free and allocate memory. The mechanism associates a reference counter with each node. Each time a process creates a pointer to a node it increments the node's reference counter atomically. When it does not intend to access a node that it has accessed before, it decrements the associated reference counter atomically. In addition to temporary links from process-local variables, each reference counter reflects the number of links in the data structure that point to the node in question. For a queue, these are the head and tail pointers and linked-list links. A node is freed only when no pointers in the data structure or temporary variables point to it.

We discovered and corrected [58] race conditions in the memory management mechanism and the associated non-blocking queue algorithm. Even so, the memory management mechanism and the queue that employs it are impractical: no finite memory can guarantee to satisfy the memory requirements of the algorithm all the time. Problems occur if a process reads a pointer to a node (incrementing the reference counter) and is then delayed. While it is not running, other processes can enqueue and dequeue an arbitrary number of additional nodes. Because of the pointer held by the delayed process, neither the node referenced by that pointer nor any of its successors can be freed. It is therefore possible to run out of memory even if the number of items in the queue is bounded by a constant. In experiments with a queue of maximum length 12 items, we ran out of memory several times during runs of ten million enqueues and dequeues, using a free list initialized with 64,000 nodes.

Most of the algorithms mentioned above are based on `compare_and_swap`, and must therefore deal with the ABA problem: if a process reads a value A in a shared location, computes a new value, and then attempts a `compare_and_swap` operation, the `compare_and_swap` may succeed when it should not, if between the read and the `compare_and_swap` some other process(es) change the A to a B and then back to an A again. The most common solution is to associate a modification counter with a pointer, to always access the counter with the pointer in any read-modify-`compare_and_swap` sequence, and to increment it in each successful `compare_and_`

swap. This solution does not guarantee that the ABA problem will not occur, but it makes it extremely unlikely. To implement this solution, one must either employ a double-word compare_and_swap, or else use array indices instead of pointers, so that they may share a single word with a counter. Valois's reference counting technique guarantees preventing the ABA problem without the need for modification counters or the double-word compare_and_swap. Mellor-Crummey's lock-free queue [55] requires no special precautions to avoid the ABA problem because it uses compare_and_swap in a fetch_and_store-modify-compare_and_swap sequence rather than the usual read-modify-compare_and_swap sequence. However, this same feature makes the algorithm blocking.

In section 2.2.2 we present two new concurrent FIFO queue algorithms inspired by ideas in the work described above. Both of the algorithms are simple and practical. One is non-blocking; the other uses a pair of locks. Correctness of these algorithms is discussed in section 2.2.3. We present experimental results in section 2.2.4. Using a 12-node SGI Challenge multiprocessor, we compare the new algorithms to a straightforward single-lock queue, Mellor-Crummey's blocking algorithm [55], and the non-blocking algorithms of Prakash *et al.* [70] and Valois [90], with both dedicated and multiprogrammed workloads. The results confirm the value of non-blocking algorithms on multiprogrammed systems. They also show consistently superior performance on the part of the new lock-free algorithm, both with and without multiprogramming. The new two-lock algorithm

cannot compete with the non-blocking alternatives on a multiprogrammed system, but outperforms a single lock when several processes compete for access simultaneously. Section 2.2.5 summarizes our conclusions.

2.2.2 The Algorithms

Figure 2.7 presents commented pseudo-code for the non-blocking queue data structure and operations. The algorithm implements the queue as a singly-linked list with *Head* and *Tail* pointers. As in Valois's algorithm, *Head* always points to a dummy node, which is the first node in the list. *Tail* points to either the last or second to last node in the list. The algorithm uses `compare_and_swap`, with modification counters to avoid the ABA problem. To allow dequeuing processes to free dequeued nodes, the dequeue operation ensures that *Tail* does not point to the dequeued node nor to any of its predecessors. This means that dequeued nodes may safely be re-used.

To obtain consistent values of various pointers we rely on sequences of reads that re-check earlier values to be sure they haven't changed. These sequences of reads are similar to, but simpler than, the snapshots of Prakash *et al.* (we need to check only one shared variable rather than two). A similar technique can be used to prevent the race condition in Stone's blocking algorithm. We use Treiber's simple and fast non-blocking stack algorithm [86] to implement a non-blocking free list.

```
record pointer_t        {ptr: pointer to node_t, count: unsigned integer}
record node_t           {value: data type, next: pointer_t}
record queue_t          {Head: pointer_t, Tail: pointer_t}


INITIALIZE(Q: pointer to queue_t)
        node := new_node()                                  # Allocate a free node
        node→next.ptr := NULL                               # Make it the only node in the linked list
        Q→Head.ptr := Q→Tail.ptr := node                    # Both Head and Tail point to it


ENQUEUE(Q: pointer to queue_t, value: data type)
E1:     node := new_node()                                  # Allocate a new node from the free list
E2:     node→value := value                                 # Copy enqueued value into node
E3:     node→next.ptr := NULL                               # Set next pointer of node to NULL
E4:     loop                                                # Keep trying until Enqueue is done
E5:         tail := Q→Tail                                  # Read Tail.ptr and Tail.count together
E6:         next := tail.ptr→next                           # Read next ptr and count fields together
E7:         if tail = Q→Tail                                # Are tail and next consistent?
E8:             if next.ptr = NULL                          # Was Tail pointing to the last node?
E9:                 if CAS(&tail.ptr→next, next, [node, next.count+1]) # Try to link node at the end of the linked list
E10:                    break                               # Enqueue is done. Exit loop
E11:                endif
E12:            else                                        # Tail was not pointing to the last node
E13:                CAS(&Q→Tail, tail, [next.ptr, tail.count+1])  # Try to swing Tail to the next node
E14:            endif
E15:        endif
E16:    endloop
E17:    CAS(&Q→Tail, tail, [node, tail.count+1])            # Try to swing Tail to the inserted node


DEQUEUE(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:     loop                                                # Keep trying until Dequeue is done
D2:         head := Q→Head                                  # Read Head
D3:         tail := Q→Tail                                  # Read Tail
D4:         next := head.ptr→next                           # Read Head.ptr→next
D5:         if head = Q→Head                                # Are head, tail, and next consistent?
D6:             if head.ptr = tail.ptr                      # Is queue empty or Tail falling behind?
D7:                 if next.ptr = NULL                      # Is queue empty?
D8:                     return FALSE                        # Queue is empty, couldn't dequeue
D9:                 endif
D10:                CAS(&Q→Tail, tail, [next.ptr, tail.count+1])  # Tail is falling behind. Try to advance it
D11:            else                                        # No need to deal with Tail
                    # Read value before CAS, otherwise another dequeue might free the next node
D12:                *pvalue := next.ptr→value
D13:                if CAS(&Q→Head, head, [next.ptr, head.count+1])  # Try to swing Head to the next node
D14:                    break                               # Dequeue is done. Exit loop
D15:                endif
D16:            endif
D17:        endif
D18:    endloop
D19:    free(head.ptr)                                      # It is safe now to free the old dummy node
D20:    return TRUE                                         # Queue was not empty, dequeue succeeded
```

Figure 2.7: Structure and operation of a non-blocking concurrent queue.

```
record node_t          {value: data type, next: pointer to node_t}
record queue_t         {Head: pointer to node_t, Tail: pointer to node_t, H_lock: lock type, T_lock: lock type}

INITIALIZE(Q: pointer to queue_t)
        node := new_node()                  # Allocate a free node
        node→next := NULL                   # Make it the only node in the linked list
        Q→Head := Q→Tail := node            # Both Head and Tail point to it
        Q→H_lock := Q→T_lock := FREE        # Locks are initially free

ENQUEUE(Q: pointer to queue_t, value: data type)
        node := new_node()                  # Allocate a new node from the free list
        node→value := value                 # Copy enqueued value into node
        node→next := NULL                   # Set next pointer of node to NULL
        lock(&Q→T_lock)                     # Acquire T_lock in order to access Tail
            Q→Tail→next := node             # Link node at the end of the linked list
            Q→Tail := node                  # Swing Tail to node
        unlock(&Q→T_lock)                   # Release T_lock

DEQUEUE(Q: pointer to queue_t, pvalue: pointer to data type): boolean
        lock(&Q→H_lock)                     # Acquire H_lock in order to access Head
            node := Q→Head                  # Read Head
            new_head := node→next           # Read next pointer
            if new_head = NULL              # Is queue empty?
                unlock(&Q→H_lock)           # Release H_lock before return
                return FALSE                # Queue was empty
            endif
            *pvalue := new_head→value       # Queue not empty. Read value before release
            Q→Head := new_head              # Swing Head to next node
        unlock(&Q→H_lock)                   # Release H_lock
        free(node)                          # Free node
        return TRUE                         # Queue was not empty, dequeue succeeded
```

Figure 2.8: Structure and operation of a two-lock concurrent queue.

Figure 2.8 presents commented pseudo-code for the two-lock queue data structure and operations. The algorithm employs separate *Head* and *Tail* locks, to allow complete concurrency between enqueues and dequeues. As in the non-blocking queue, we keep a dummy node at the beginning of the list. Because of the dummy node, enqueuers never have to access *Head*, and dequeuers never have to access *Tail*, thus avoiding potential deadlock problems that arise from processes trying to acquire the locks in different orders.

2.2.3 Correctness

Safety

The presented algorithms are safe because they satisfy the following properties:

1. The linked list is always connected.
2. Nodes are only inserted after the last node in the linked list.
3. Nodes are only deleted from the beginning of the linked list.
4. *Head* always points to the first node in the linked list.
5. *Tail* always points to a node in the linked list.

Initially, all these properties hold. By induction, we show that they continue to hold, assuming that the ABA problem never occurs.

1. The linked list is always connected because once a node is inserted, its *next* pointer is not set to NULL before it is freed, and no node is freed until it is deleted from the beginning of the list (property 3).
2. In the lock-free algorithm, nodes are only inserted at the end of the linked list because they are linked through the *Tail* pointer, which always points to a node in the linked-list (property 5), and an inserted node is linked only to a node that has a NULL *next* pointer, and the only such node in the linked list is the last one (property 1).

In the lock-based algorithm nodes are only inserted at the end of the linked list because they are inserted after the node pointed to by *Tail*, and in this algorithm *Tail* always points to the last node in the linked list, unless it is protected by the tail lock.

3. Nodes are deleted from the beginning of the list, because they are deleted only when they are pointed to by *Head* and *Head* always points to the first node in the list (property 4).

4. *Head* always points to the first node in the list, because it only changes its value to the next node atomically (either using the head lock or using `compare_and_swap`). When this happens the node it used to point to is considered deleted from the list. The new value of *Head* cannot be NULL because if there is one node in the linked list the dequeue operation returns without deleting any nodes.

5. *Tail* always points to a node in the linked list, because it never lags behind *Head*, so it can never point to a deleted node. Also, when *Tail* changes its value it always swings to the next node in the list and it never tries to change its value if the *next* pointer is NULL.

Linearizability

The presented algorithms are linearizable because there is a specific point during each operation at which it is considered to "take effect" [28]. An enqueue takes effect when the allocated node is linked to the last node in the linked list. A dequeue takes effect when *Head* swings to the next node. And, as shown in the previous subsection (properties 1, 4, and 5), the queue variables always reflect the state of the queue; they never enter a transient state in which the state of the queue can be mistaken (e.g. a non-empty queue appears to be empty).

Liveness

The Lock-Free Algorithm is Non-Blocking

The lock-free algorithm is non-blocking because if there are non-delayed processes attempting to perform operations on the queue, an operation is guaranteed to complete within finite time.

An enqueue operation loops only if the condition in line E7 fails, the condition in line E8 fails, or the `compare_and_swap` in line E9 fails. A dequeue operation loops only if the condition in line D5 fails, the condition in line D6 holds (and the queue is not empty), or the `compare_and_swap` in line D13 fails.

We show that the algorithm is non-blocking by showing that a process loops beyond a finite number of times only if another process completes an operation

on the queue.

- The condition in line E7 fails only if *Tail* is written by an intervening process after executing line E5. *Tail* always points to the last or second to last node of the linked list, and when modified it follows the *next* pointer of the node it points to. Therefore, if the condition in line E7 fails more than once, then another process must have succeeded in completing an enqueue operation.

- The condition in line E8 fails if *Tail* was pointing to the second to last node in the linked-list. After the `compare_and_swap` in line E13, *Tail* must point to the last node in the list, unless a process has succeeded in enqueuing a new item. Therefore, if the condition in line E8 fails more than once, then another process must have succeeded in completing an enqueue operation.

- The `compare_and_swap` in line E9 fails only if another process succeeded in enqueuing a new item to the queue.

- The condition in line D5 and the `compare_and_swap` in line D13 fail only if *Head* has been written by another process. *Head* is written only when a process succeeds in dequeuing an item.

- The condition in line D6 succeeds (while the queue is not empty) only if *Tail* points to the second to last node in the linked list (in this case it is also the first node). After the `compare_and_swap` in line D10, *Tail* must point to the last node in the list, unless a process succeeded in enqueuing a new

item. Therefore, if the condition of line D6 succeeds more than once, then another process must have succeeded in completing an enqueue operation (and the same or another process succeeded in dequeuing an item).

The Two-Lock Algorithm is Livelock-Free

The two-lock algorithm does not contain any loops. Therefore, if the mutual exclusion lock algorithm used for locking and unlocking the head and tail locks is livelock-free, then the presented algorithm is livelock-free too. There are many mutual exclusion algorithms that are livelock-free [56].

2.2.4 Performance

We use a 12-processor Silicon Graphics Challenge multiprocessor to compare the performance of the new algorithms to that of a single-lock algorithm, the algorithm of Prakash *et al.* [70], Valois's algorithm [90] (with corrections to the memory management mechanism [58]), and Mellor-Crummey's algorithm [55]. We include the algorithm of Prakash *et al.* because it appears to be the best of the known non-blocking alternatives. Mellor-Crummey's algorithm represents non-lock-based but blocking alternatives; it is simpler than the code of Prakash *et al.*, and could be expected to display lower constant overhead in the absence of unpredictable process delays, but is likely to degenerate on a multiprogrammed system. We include Valois's algorithm to demonstrate that on multiprogrammed

systems even a comparatively slow non-blocking algorithm can outperform blocking algorithms.

For the two lock-based algorithms we use test-and-`test_and_set` locks with bounded exponential backoff [5; 56]. We also use backoff where appropriate in the non-lock-based algorithms. Performance was not sensitive to the exact choice of backoff parameters in programs that do at least a modest amount of work between queue operations. We emulate both `test_and_set` and the atomic operations required by the other algorithms (`compare_and_swap`, `fetch_and_increment`, `fetch_and_decrement`, etc.) using the MIPS R4000 `load_linked` and `store_conditional` instructions.[5]

To ensure the accuracy of the experimental results, we used the multiprocessor exclusively and prevented other users from accessing it during the experiments. To evaluate the performance of the algorithms under different levels of multiprogramming, we used a feature in the Irix operating system that allows programmers to associate processes with certain processors. For example, to represent a dedicated system on which multiprogramming is not permitted, we created as many processes as the number of processors we wanted to use and locked each process to a different processor. And in order to represent a system with a multiprogram-

[5] `Load_linked` and `store_conditional`, proposed by Jensen *et al.* [36], must be used together to read, modify, and write a shared location. `Load_linked` returns the value stored at the shared location. `Store_conditional` checks if any other processor has since written to that location. If not then the location is updated and the operation returns success, otherwise it returns failure. `Load_linked` and `store_conditional` are supported by the MIPS II, PowerPC, and Alpha architectures.

ming level of 2, we created twice as many processes as the number of processors we wanted to use, and locked each pair of processes to an individual processor.

The algorithms were compiled at the highest optimization level, and were carefully hand-optimized. We tested each of the algorithms in hours-long executions on various numbers of processors. It was during this process that we discovered the race conditions mentioned in section 2.2.1.

All the experiments employ an initially-empty queue to which processes perform a series of enqueue and dequeue operations. Each process enqueues an item, does "other work", dequeues an item, does "other work", and repeats. With p processes, each process executes this loop $\lfloor 10^6/p \rfloor$ or $\lceil 10^6/p \rceil$ times, for a total of one million enqueues and dequeues. The "other work" consists of approximately 6 μs of spinning in an empty loop; it serves to make the experiments more realistic by preventing long runs of queue operations by the same process (which would display overly-optimistic performance due to an unrealistically low cache miss rate). We subtracted the time required for one processor to complete the "other work" from the total time reported in the figures.

Figure 2.9 shows net elapsed time in seconds for one million enqueue/dequeue pairs. Roughly speaking, this corresponds to the time in microseconds for one enqueue/dequeue pair. More precisely, for k processors, the graph shows the time one processor spends performing $10^6/k$ enqueue/dequeue pairs, plus the amount by which the critical path of the other $10^6(k-1)/k$ pairs performed by other

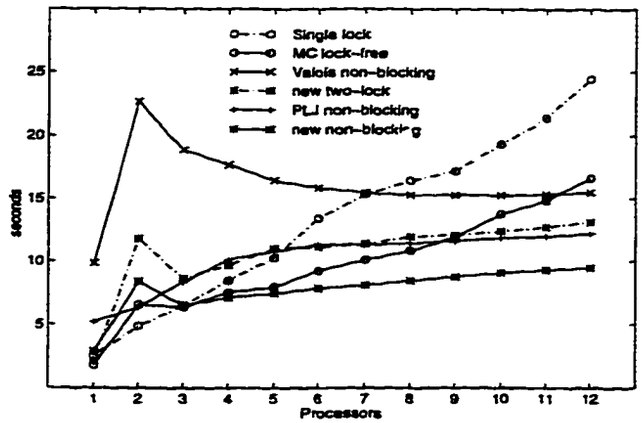

Figure 2.9: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

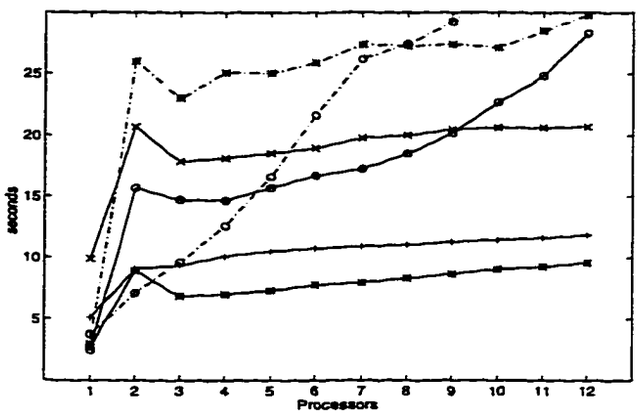

Figure 2.10: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 2 processes per processor.

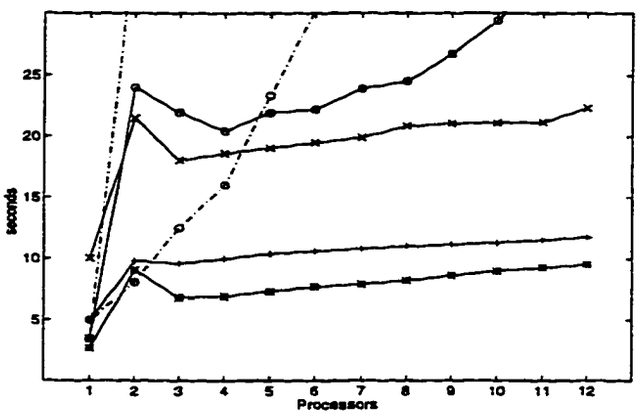

Figure 2.11: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 3 processes per processor.

processors exceeds the time spent by the first processor in "other work" and loop overhead. For $k = 1$, the second term is zero. As k increases, the first term shrinks toward zero, and the second term approaches the critical path length of the overall computation; i.e. one million times the *serial portion* of an enqueue/dequeue pair. Exactly how much execution will overlap in different processors depends on the choice of algorithm, the number of processors k, and the length of the "other work" between queue operations.

With only one processor, memory references in all but the first loop iteration hit in the cache, and completion times are very low. With two processors active, contention for head and tail pointers and queue elements causes a high fraction of references to miss in the cache, leading to substantially higher completion times. The queue operations of processor 2, however, fit into the "other work" time of processor 1, and vice versa, so we are effectively measuring the time for one processor to complete 5×10^5 enqueue/dequeue pairs. At three processors, the cache miss rate is about the same as it was with two processors. Each processor only has to perform $10^6/3$ enqueue/dequeue pairs, but some of the operations of the other processors no longer fit in the first processor's "other work" time. Total elapsed time decreases, but by a fraction less than 1/3. Toward the right-hand side of the graph, execution time rises for most algorithms as smaller and smaller amounts of per-processor "other work" and loop overhead are subtracted from a total time dominated by critical path length. In the single-lock and Mellor-

Crummey curves, the increase is probably accelerated as high rates of contention increase the average cost of a cache miss. In Valois's algorithm, the plotted time continues to decrease, as more and more of the memory management overhead moves out of the critical path and into the overlapped part of the computation.

Figures 2.10 and 2.11 plot the same quantity as Figure 2.9, but for a system with 2 and 3 processes per processor, respectively. The operating system multiplexes the processor among processes with a scheduling quantum of 10 ms. As expected, the blocking algorithms fare much worse in the presence of multiprogramming, since an inopportune preemption can block the progress of every process in the system. Also as expected, the degree of performance degradation increases with the level of multiprogramming.

In all three graphs, the new non-blocking queue outperforms all of the other alternatives when three or more processors are active. Even for one or two processors, its performance is good enough that we can comfortably recommend its use in all situations. The two-lock algorithm outperforms the one-lock algorithm when more than 5 processors are active on a dedicated system: it appears to be a reasonable choice for machines that are not multiprogrammed, and that lack a universal[6] atomic primitive (`compare_and_swap` or `load_linked/store_conditional`).

[6]Herlihy [29] presented a hierarchy of non-blocking objects that also applies to atomic primitives. A primitive is at level n of the hierarchy if it can provide a non-blocking solution to a consensus problem for up to n processors. Primitives at higher levels of the hierarchy can provide non-blocking implementations of those at lower levels, but not conversely. `Compare_and_swap` and the pair `load_linked` and `store_conditional` are *universal* primitives as they are at level ∞ of the hierarchy. Widely supported primitives such as `test_and_set`, `fetch_and_add`, and `fetch_and_store` are at level 2.

2.2.5 Summary

Queues are ubiquitous in parallel programs, and their performance is a matter of major concern. We have presented a concurrent queue algorithm that is simple, non-blocking, practical, and fast. We were surprised not to find it in the literature. It seems to be the algorithm of choice for any queue-based application on a multiprocessor with universal atomic primitives (e.g. `compare_and_swap` or `load_linked/store_conditional`).

We have also presented a queue with separate head and tail pointer locks. Its structure is similar to that of the non-blocking queue, but it allows only one enqueue and one dequeue to proceed at a given time. Because it is based on locks, however, it will work on machines with such simple atomic primitives as `test_and_set`. We recommend it for heavily-utilized queues on such machines (For a queue that is usually accessed by only one or two processors, a single lock will run a little faster.)

Immunity to arbitrary processes delays is a primary benefit of non-blocking parallel algorithms and preemption-safe locking. In the next chapter, we compare the performance of these two approaches in the context of multiprogrammed systems for several important data structures, including queues.

3 Synchronization and Multiprogramming

3.1 Introduction

In order to achieve acceptable response time and high utilization, most multiprocessors are multiprogrammed by time-slicing processors among processes. The performance of mutual exclusion locks in parallel applications degrades significantly on time-slicing multiprogrammed systems [94] due to the preemption of processes holding locks. Any other processes busy-waiting on the lock are then unable to perform useful work until the preempted process is rescheduled and subsequently releases the lock.

Alternative multiprogramming schemes to time-slicing have been proposed in order to avoid the adverse effect of time-slicing on the performance of synchronization operations. However, each has limited applicability and/or reduces the utilization of the multiprocessor. Coscheduling [67], ensures that all processes of

an application run together. It has the disadvantage of reducing the utilization of the multiprocessor if applications have a variable amount of parallelism, or if processes cannot be evenly assigned to time-slices of the multiprocessor. Another alternative is hardware partitioning [87], under which no two applications share a processor. However, fixed size partitions have the disadvantage of resulting in poor response time when the number of processes is larger than the number of processors, and adjustable size partitions have the disadvantage of requiring applications to be able to adjust their number of processes as new applications join the system. Otherwise, processes from the same application might have to share the same processor, allowing one to be preempted while holding a mutual exclusion lock. Traditional time-slicing remains the most widely used scheme of multiprogramming on multiprocessor systems.

For time-sliced systems, researchers have proposed two principal strategies to avoid inopportune preemption: *preemption safe locking* and *non-blocking algorithms*. Most preemption-safe locking techniques require a widening of the kernel interface, to facilitate cooperation between the application and the kernel. Generally, these techniques try either to recover from the preemption of lock-holding processes (or processes waiting on queued locks), or to avoid preempting processes while holding locks.

An implementation of a data structure is *non-blocking* (also known as *lock-free*) if it guarantees that at least one process of those trying to concurrently update the

data structure will succeed in completing its operation within a bounded amount of time, assuming that at least one process is active, regardless of the state of other processes. Non-blocking algorithms do not require any communication with the kernel and by definition they cannot use mutual exclusion. Rather, they generally rely on hardware support for a universal atomic primitive such as `compare_and_swap` or the pair `load_linked` and `store_conditional`, while mutual exclusion locks can be implemented using weaker atomic primitives such as `test_and_set`, `fetch_and_increment`, or `fetch_and_store`.

Few of the above mentioned techniques have been evaluated experimentally, and then only in comparison to ordinary (preemption-oblivious) mutual exclusion locks. We evaluate the relative performance of preemption-safe and non-blocking atomic update techniques on multiprogrammed (time-sliced) as well as dedicated multiprocessor systems. We focus on four important data structures: queues, stacks, heaps, and counters. Our experimental results, employing both micro-benchmarks and real applications, on a 12-processor Silicon Graphics Challenge multiprocessor, indicate that fast data-structure-specific non-blocking algorithms outperform both ordinary and preemption-safe lock-based alternatives, not only on time-sliced systems, but on dedicated machines as well [62]. At the same time, preemption-safe algorithms outperform ordinary locks on time-sliced systems, and should therefore be supported by multiprocessor operating systems. We do not examine general-purpose non-blocking techniques in detail; previous work indicates

that they are highly inefficient, though they provide a level of fault tolerance unavailable with locks.

The rest of this chapter is organized as follows. We discuss preemption-safe locking in Section 3.2, and non-blocking algorithms in Section 3.3. We describe our experimental methodology and results in Section 3.4. Finally, we summarize our conclusions and recommendations in Section 3.5.

3.2 Preemption-Safe Locking

For simple mutual exclusion locks (e.g. `test_and_set`), preemption-safe locking techniques allow the system either to avoid or to recover from the adverse effect of the preemption of processes holding locks. Edler *et al.*'s Symunix system [15] employs an avoidance technique: a process may set a flag requesting that the kernel not preempt it because it is holding a lock. The kernel will honor the request up to a pre-defined time limit, setting a second flag to indicate that it did so, and deducting any extra execution time from the beginning of the process's next quantum. A process should yield the processor if it finds, upon leaving a critical section, that it was granted an extension.

The *first-class threads* of Marsh *et al.*'s Psyche system [53] employ a different avoidance technique: they require the kernel to warn an application process a fixed amount of time in advance of preemption, by setting a flag that is visible in user

space. If a process verifies that the flag is unset before entering a critical section (and if critical sections are short), then it is guaranteed to be able to complete its operation in the current quantum. If it finds the flag is set, it can voluntarily yield the processor.

Recovery-based preemption-safe locking techniques include the *spin-then-block* locks of Ousterhout [67] which let a waiting process spin for a certain period of time and then—if unsuccessful in entering the critical section—block, thus minimizing the adverse effect of waiting for a lock held by a descheduled process. Karlin *et al.* [39] present a set of spin-then-block alternatives that adjust the spin time based on past experience. Black's work on Mach [11] introduced another recovery technique: a process may suggest to the kernel that it be descheduled in favor of some specific other process (presumably the one that is holding a desired lock). The *scheduler activations* of Anderson *et al.* [7] also support recovery: when a processor is taken from an application, another processor belonging to the same application is informed via software interrupt. If the preempted process was holding a lock, the interrupted processor can perform a context switch to the preempted process and push it through the critical section.

Simple preemption-safe techniques rely on the fact that processes acquire a `test_and_set` lock in non-deterministic order. Unfortunately, `test_and_set` locks do not scale well to large machines. Queue-based locks scale well, but impose a deterministic order on lock acquisitions, forcing a preemption-safe technique to

deal with preemption not only of the process holding a lock, but of processes waiting in the lock's queue as well. Preempting and scheduling processes in an order inconsistent with their order in the lock's queue can degrade performance dramatically. Kontothanassis *et al.* [41] present preemption-safe (or "scheduler-conscious") versions of the ticket lock, the MCS lock [56], and Krieger *et al.*'s reader-writer lock [42]. These algorithms detect the descheduling of critical processes using handshaking and/or a widened kernel-user interface, and use this information to avoid handing the lock to a preempted process.

The proposals of Black and of Anderson *et al.* require the application to recognize the preemption of lock-holding processes and to deal with the problem. By performing recovery on a processor other than the one on which the preempted process last ran, they also sacrifice cache footprint. The proposal of Marsh *et al.* requires the application to estimate the maximum duration of a critical section, which is not always possible. To represent the preemption-safe approach in our experiments, we employ test-and-`test_and_set` locks with exponential backoff, based on the kernel interface of Edler *et al.* For machines the size of ours (12 processors), the results of Kontothanassis *et al.* indicate that these will outperform queue-based locks.

3.3 Non-Blocking Algorithms

Several non-blocking implementations of widely used data structures as well as general methodologies for developing such implementations systematically have been proposed in the literature. These implementations and methodologies were motivated in large part by the performance degradation of mutual exclusion locks as a result of arbitrary process delays, particularly those due to preemption on a multiprogrammed system.

3.3.1 General Non-Blocking Methodologies

Herlihy [30] presented a general methodology for transforming sequential implementations of data structures into concurrent non-blocking implementations using `compare_and_swap` or `load_linked/store_conditional`. The basic methodology requires copying the entire data structure on every update. Herlihy also proposed an optimization by which the programmer can avoid some fraction of the copying for certain data structures; he illustrated this optimization in a non-blocking implementation of a skew-heap-based priority queue. Alemany and Felten [3] and LaMarca [45] proposed techniques to reduce unnecessary copying and useless parallelism associated with Herlihy's methodologies using extra communication between the operating system kernel and application processes. Barnes [9] presented a general methodology in which processes record and times-

tamp their modifications to the shared object, and cooperate whenever conflicts arise. Shavit and Touitou [77] presented *software transactional memory*, which implements a k-word `compare_and_swap` using `load_linked/store_conditional`. Also, Anderson and Moir [8] presented non-blocking methodologies for large objects that rely on techniques for implementing multiple-word `compare_and_swap` using `load_linked/store_conditional` and vice versa. Turek *et al.* [88] and Prakash *et al.* [69] presented methodologies for transforming multiple lock concurrent objects into lock-free concurrent objects. Unfortunately, the performance of non-blocking algorithms resulting from general methodologies is acknowledged to be significantly inferior to that of the corresponding lock-based algorithms [30; 45; 77].

Two proposals for hardware support for general non-blocking data structures have been presented: *transactional memory* by Herlihy and Moss [31] and the *Oklahoma update* by Stone *et al.* [85]. Neither of these techniques has been implemented on a real machine. The simulation-based experimental results of Herlihy and Moss show performance significantly inferior to that of spin locks. Stone *et al.* did not present experimental results.

3.3.2 Data-Structure-Specific Non-Blocking Algorithms

Treiber [86] proposed a non-blocking implementation of concurrent link-based stacks. It represents the stack as a singly-linked list with a *Top* pointer. It uses

```
record pointer_t          {ptr: pointer to node_t, count: unsigned integer}
record node_t             {value: data type, next: pointer_t}
record stack_t            {Top: pointer_t}

INITIALIZE(S: pointer to stack_t)
        S→Top.ptr := NULL                                  # Empty stack. Top points to NULL

PUSH(S: pointer to stack_t, value: data type)
        node := new_node()                                 # Allocate a new node from the free list
        node→value := value                                # Copy stacked value into node
        repeat                                             # Keep trying until Push is done
            top := S→Top                                   # Read Top.ptr and Top.count together
            node→next.ptr := top.ptr                       # Link new node to head of list
        until CAS(&S→Top, top, [node, top.count+1])        # Try to swing Top to new node

POP(S: pointer to stack_t, pvalue: pointer to data type): boolean
        repeat                                             # Keep trying until Pop is done
            top := S→Top                                   # Read Top
            if top.ptr = NULL                              # Is the stack empty?
                return FALSE                               # The stack was empty, couldn't pop
            endif
        until CAS(&S→Top, top, [top.ptr→next.ptr, top.count+1])   # Try to swing Top to the next node
        *pvalue := top.ptr→value                           # Pop is done. Read value
        free(top.ptr)                                      # It is safe now to free the old node
        return TRUE                                        # The stack was not empty, pop succeeded
```

Figure 3.1: Structure and operation of Treiber's non-blocking concurrent stack algorithm [86].

`compare_and_swap` to modify the value of *Top* atomically. Commented pseudo-code of Treiber's non-blocking stack algorithm is presented in Figure 3.1. No performance results were reported for non-blocking stacks. However, Treiber's stack is very simple and can be expected to be quite fast. We also observe that a stack derived from Herlihy's general methodology, with unnecessary copying removed, seems to be simple enough to compete with lock-based algorithms.

Valois [91] proposed a non-blocking implementation of linked lists. Anderson and Woll [6] proposed a non-blocking solution to the union-find problem. Simple non-blocking centralized counters can be implemented trivially using a `fetch_and_add` atomic primitive (if supported by hardware), or a read–modify–check–

```
ADD(X: pointer to integer, value: integer): integer
        repeat                                          # Keep trying until SC succeeds
            count := LL(X)                              # Read the current value of X
        until SC(X, count+value)
        return count                                    # Add is done, return previous value
```

Figure 3.2: A non-blocking concurrent counter using `load-linked` and `store-conditional`.

write cycle using `compare_and_swap` or `load_linked/store_conditional`. Figure 3.2 shows a non-blocking counter implementation using `load_linked/store_conditional`.

Massalin and Pu [54] presented non-blocking algorithms for array-based stacks, array-based queues, and linked lists. Unfortunately, as mentioned in Section 2.2.1, their algorithms require `double_compare_and_swap`, a primitive that operates on two arbitrary memory locations simultaneously, and that appears to be available only on the Motorola 68020 processor and its direct descendants.[1] No practical non-blocking implementations for array-based stacks or circular queues have been proposed. The general methodologies can be used, but the resulting algorithms would be very slow. For these data structures lock-based algorithms seem to be the only option.

Data-structure-specific non-blocking queue algorithms were discussed in detail in Section 2.2.1.

[1] Greenwald and Cheriton use simulation for evaluating the performance of a linked-list implementation based on `double_compare_and_swap` [22].

3.4 Experimental Results

We use a Silicon Graphics Challenge multiprocessor with twelve 100 MHz MIPS R4000 processors to compare the performance of the most promising non-blocking, ordinary lock-based, and preemption-safe lock-based implementations of counters and of link-based queues, stacks, and skew heaps. We use micro-benchmarks to compare the performance of the alternative algorithms under various levels of contention. We also use two versions of a parallel quicksort application, together with a parallel solution to the traveling salesman problem, to compare the performance of the algorithms when used in a real application.

To ensure the accuracy of our results regarding the level of multiprogramming, we prevented other users from accessing the multiprocessor during the experiments. To evaluate the performance of the algorithms under different levels of multiprogramming, we used a feature of the Challenge's Irix operating system that allows programmers to pin processes to processors. We then used one of the processors to serve as a pseudo-scheduler. Whenever a process is due for preemption, the pseudo-scheduler interrupts it, forcing it into a signal handler. The handler spins on a flag which the pseudo-scheduler sets when the process can continue computation. The time spent executing the handler represents the time during which the processor is taken from the process and handed over to a process that belongs to some other application. The time quantum is 10 ms.

All ordinary and preemption-safe locks used in the experiments are test-and-

```
TestAndSet(X: pointer to boolean): boolean
        repeat                          # Keep trying SC succeeds or X is TRUE
            local := LL(X)              # Read the current value of X
            if local = TRUE
                return TRUE             # TAS should return TRUE
        until SC(X, TRUE)
        return FALSE                    # TAS is done, indicate that X was FALSE

CompareAndSwap(X: pointer to integer, expected: integer, new: integer): boolean
        repeat                          # Keep trying until SC succeeds or X ≠ expected
            local := LL(X)              # Read the current value of X
            if local ≠ expected
                return FALSE            # CAS should fail
        until SC(X, new)
        return TRUE                     # CAS succeeded
```

Figure 3.3: Implementations of `test-and-set` and `compare-and-swap` using `load-linked` and `store-conditional`.

`test_and_set` locks with bounded exponential backoff. All non-blocking algorithms also use bounded exponential backoff. The effectiveness of backoff in reducing contention on locks and synchronization data is demonstrated in the literature [5; 56]. The backoff was chosen to yield good overall performance for all algorithms, and not to exceed 30 μs. We emulate both `test_and_set` and `compare_and_swap`, using `load_linked` and `store_conditional` instructions, as shown in Figure 3.3.

In the figures, multiprogramming level represents the number of applications sharing the machine, with one process per processor per application. A multiprogramming level of 1 (the top graph in each figure) therefore represents a dedicated machine; a multiprogramming level of 3 (the bottom graph in each figure) represents a system with a process from each of three different applications on each processor.

3.4.1 Queues

Figure 3.4 shows performance results for eight queue implementations on a dedicated system (no multiprogramming), and on multiprogrammed systems with 2 and 3 processes per processor. The eight implementations are: the usual single-lock algorithm using both ordinary and preemption-safe locks (**single ordinary lock** and **single safe lock**); our two-lock algorithm, again using both ordinary and preemption-safe locks (**two ordinary locks** and **two safe locks**); our non-blocking algorithm (**MS non-blocking**) and those due to Prakash *et al.* [70] (**PLJ non-blocking**) and Valois [89] (**Valois non-blocking**); and Mellor-Crummey's blocking algorithm [55] (**MC blocking**). We include the algorithm of Prakash *et al.* because it appears to be the best of the known non-blocking alternatives. Mellor-Crummey's algorithm represents non-lock-based but blocking alternatives; it is simpler than the code of Prakash *et al.*, and could be expected to display lower constant overhead in the absence of unpredictable process delays, but is likely to degenerate on a multiprogrammed system. We include Valois's algorithm to demonstrate that on multiprogrammed systems even a comparatively slow non-blocking algorithm can outperform blocking algorithms.

The horizontal axes of the graphs represent the number of processors. The vertical axes represent execution time normalized to that of the preemption-safe single lock algorithm. This algorithm was chosen as the basis of normalization because it yields the median performance among the set of algorithms. We use

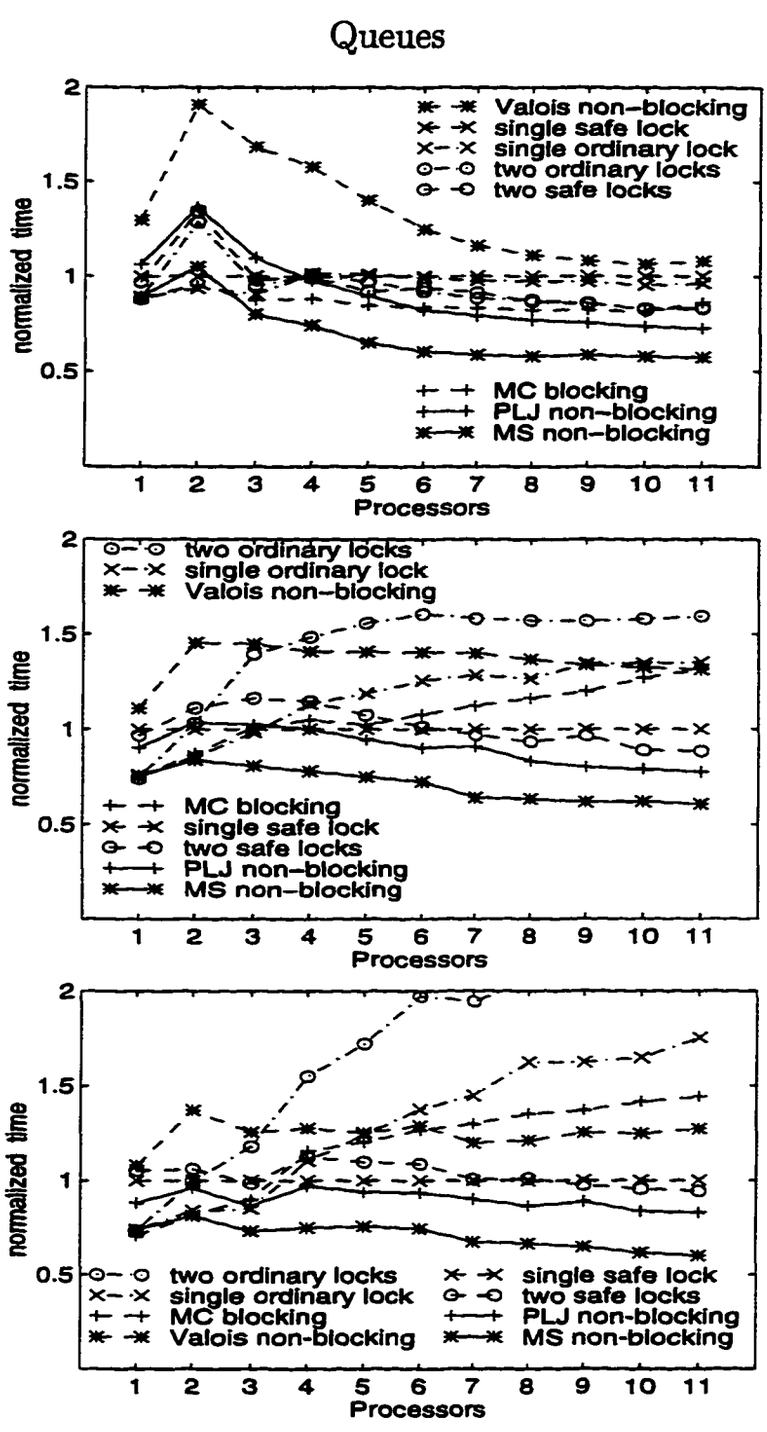

Figure 3.4: Normalized execution time for one million enqueue/dequeue pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

normalized time in order to show the difference in performance between the algorithms uniformly across different numbers of processors. If we were to use absolute time, the vertical axes would have to be extended to cover the high absolute execution time on a single processor, making the graph too small to read for larger numbers of processors. The absolute times in seconds for the preemption-safe single-lock algorithm on one and 11 processors, with 1, 2, and 3 processes per processor, are 18.2 and 15.6, 38.8 and 15.4, and 57.6 and 16.3, respectively.

The execution time is the time taken by all processors to perform one million pairs of enqueues and dequeues to an initially empty queue (each process performs 1,000,000/p enqueue/dequeue pairs, where p is the number of processors). Every process spends 6 μs ($\pm$10% randomization) spinning in an empty loop after performing every enqueue or dequeue operation (for a total of 12 μs per iteration). This time is meant to represent "real" computation. It prevents one process from dominating the data structure and finishing all its operations while other processes are starved by caching effects and backoff.

The results show that as the level of multiprogramming increases, the performance of ordinary locks and Mellor-Crummey's blocking algorithm degrades significantly, while the performance of preemption-safe locks and non-blocking algorithms remains relatively unchanged. The "bump" at two processors is due primarily to cache misses, which do not occur on one processor, and to a smaller amount of overlapped computation, in comparison to larger numbers of proces-

QUEUES

Single ordinary lock	16.2
MC blocking	16.2
MS non-blocking	16.3
Two ordinary locks	16.9
Two safe locks	17.7
Single safe lock	18.2
PLJ non-blocking	19.4
Valois non-blocking	23.7

Table 3.1: Execution times in seconds for one million enqueue/dequeue pairs on a single processor (no contention).

sors. This effect is more obvious in the multiple lock and non-blocking algorithms, which have a greater potential amount of overlap among concurrent operations.

The two-lock algorithm outperforms the single-lock in the case of high contention since it allows more concurrency, but it suffers more with multiprogramming when using ordinary locks, as the chances are larger that a process will be preempted while holding a lock needed by other processes. On a dedicated system, the two-lock algorithm outperforms a single lock when more than 4 processors are active in our micro-benchmark. With multiprogramming levels of 2 and 3, the cross-over points for the one and two-lock algorithms with preemption-safe locks occur at 6 and 8 processors, respectively. The non-blocking algorithms, except for that of Valois, provide better performance; they enjoy added concurrency without the overhead of extra locks, and without being vulnerable to interference from multiprogramming. Valois's algorithm suffers from the high overhead of the complex memory management technique associated with it.

Table 3.1 shows absolute execution times for the eight queue implementations

STACKS		
	Treiber non-blocking	15.4
	Ordinary lock	15.8
	Herlihy non-blocking	16.4
	Preemption-safe lock	19.0

Table 3.2: Execution times in seconds for one million push/pop pairs on a single processor (no contention).

on a single processor. (They correspond to the left-most points in the top graph of Figure 3.4.) In the absence of contention, any overhead required to communicate with the scheduler in a preemption-safe algorithm is "wasted", but the numbers indicate that this overhead is low.

Overall, our non-blocking algorithm yields the best performance. It outperforms the single-lock preemption-safe algorithm by more than 40% on 11 processors with various levels of multiprogramming, since it allows more concurrency and needs to access fewer memory locations. In the case of no contention, it is essentially tied with the single ordinary lock and with Mellor-Crummey's queue.

3.4.2 Stacks

Figure 3.5 shows performance results for four stack implementations on a dedicated system, and on multiprogrammed systems with 2 and 3 processes per processor. Table 3.2 shows performance on a dedicated processor—the left-most points in the top-most graph. The four stack implementations are: the usual single lock algorithm using ordinary and preemption-safe locks, Treiber's non-blocking

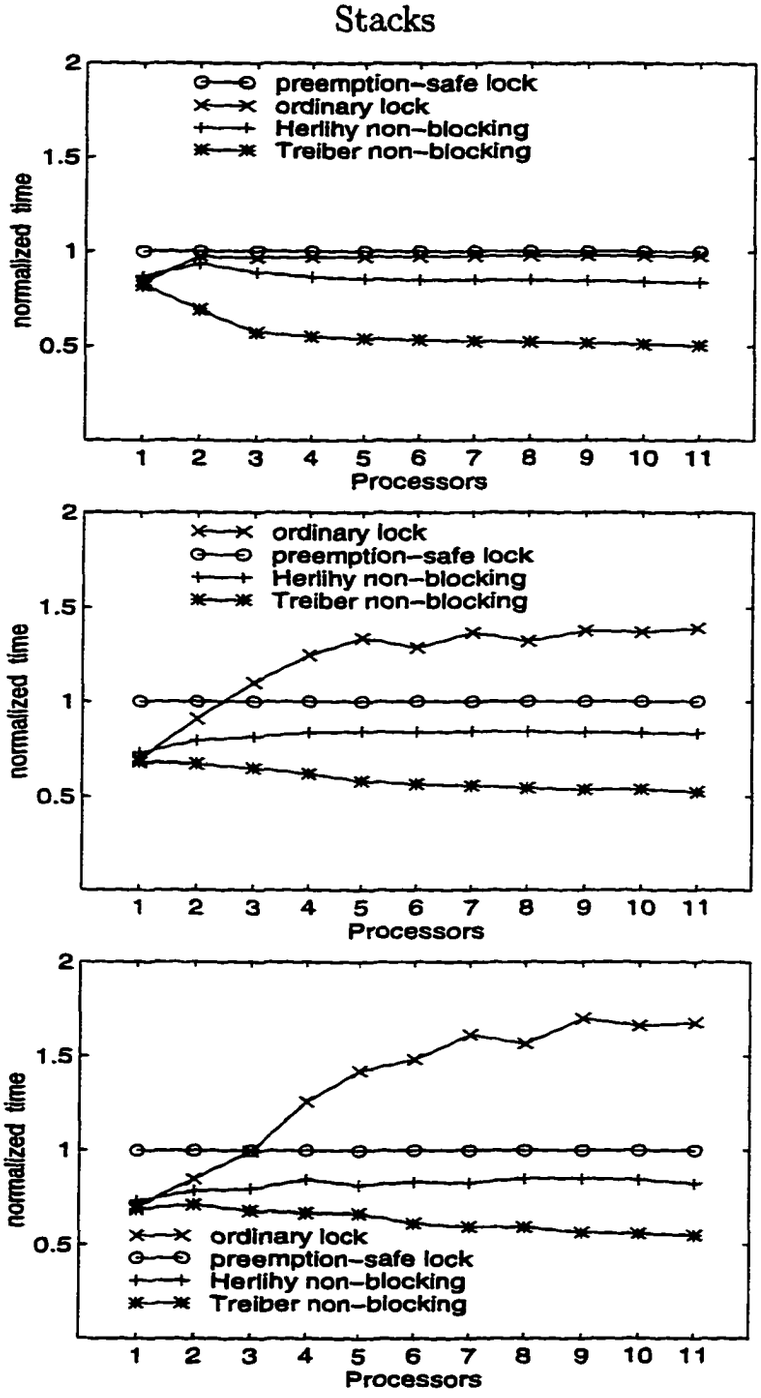

Figure 3.5: Normalized execution time for one million push/pop pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

stack algorithm [86], and an optimized non-blocking algorithm based on Herlihy's general methodology [30].

Like Treiber's non-blocking stack algorithm, the optimized algorithm based on Herlihy's methodology uses a singly-linked list to represent the stack with a *Top* pointer. However, every process has its own copy of *Top* and an operation is successfully completed only when the process uses `load_linked/store_conditional` to swing a shared pointer to its copy of *Top*. The shared pointer can be considered as pointing to the latest version of the stack.

The axes in the graphs have the same semantics as those in the queue graphs. Execution time is normalized to that of the preemption-safe single lock algorithm. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with 1, 2, and 3 processes per processor, are 19.0 and 20.3, 40.8 and 20.7, and 60.2 and 21.6, respectively. Each process executes 1,000,000/p push/pop pairs on an initially empty stack, with a 6 μs average delay between successive operations.

As the level of multiprogramming increases, the performance of ordinary locks degrades, while the performance of the preemption-safe and non-blocking algorithms remains relatively unchanged. Treiber's algorithm outperforms all the others even on dedicated systems. It outperforms the preemption-safe algorithm by over 45% on 11 processors with various levels of multiprogramming. This is mainly due to the fact that a push or a pop in Treiber's algorithm typically needs

HEAPS	Ordinary lock	20.4
	Preemption-safe lock	21.0
	Herlihy non-blocking	22.1

Table 3.3: Execution times in seconds for one million insert/delete_min pairs on a single processor (no contention).

to access only two cache lines in the data structure, while a lock-based algorithm has the overhead of accessing lock variables as well. Accordingly, Treiber's algorithm yields the best performance even with no contention.

3.4.3 Heaps

Figure 3.6 shows performance results for three skew heap implementations[2] on a dedicated system, and on multiprogrammed systems with 2 and 3 processes per processor. Table 3.3 shows performance on a dedicated processor. The three implementations are: the usual single-lock algorithm using ordinary and preemption-safe locks, and an optimized non-blocking algorithm due to Herlihy [30].

The optimized non-blocking algorithm due to Herlihy uses a binary tree to represent the heap with a *Root* pointer. Every process has its own copy of *Root*. A process performing a heap operation copies the nodes it intends to modify to local free nodes and finally tries to swing a global shared pointer to its copy of *Root* using `load_linked/store_conditional`. If it succeeds, the local copies of

[2]Our heap implementation in Section 2.1 is array-based and is not applicable to skew heaps.

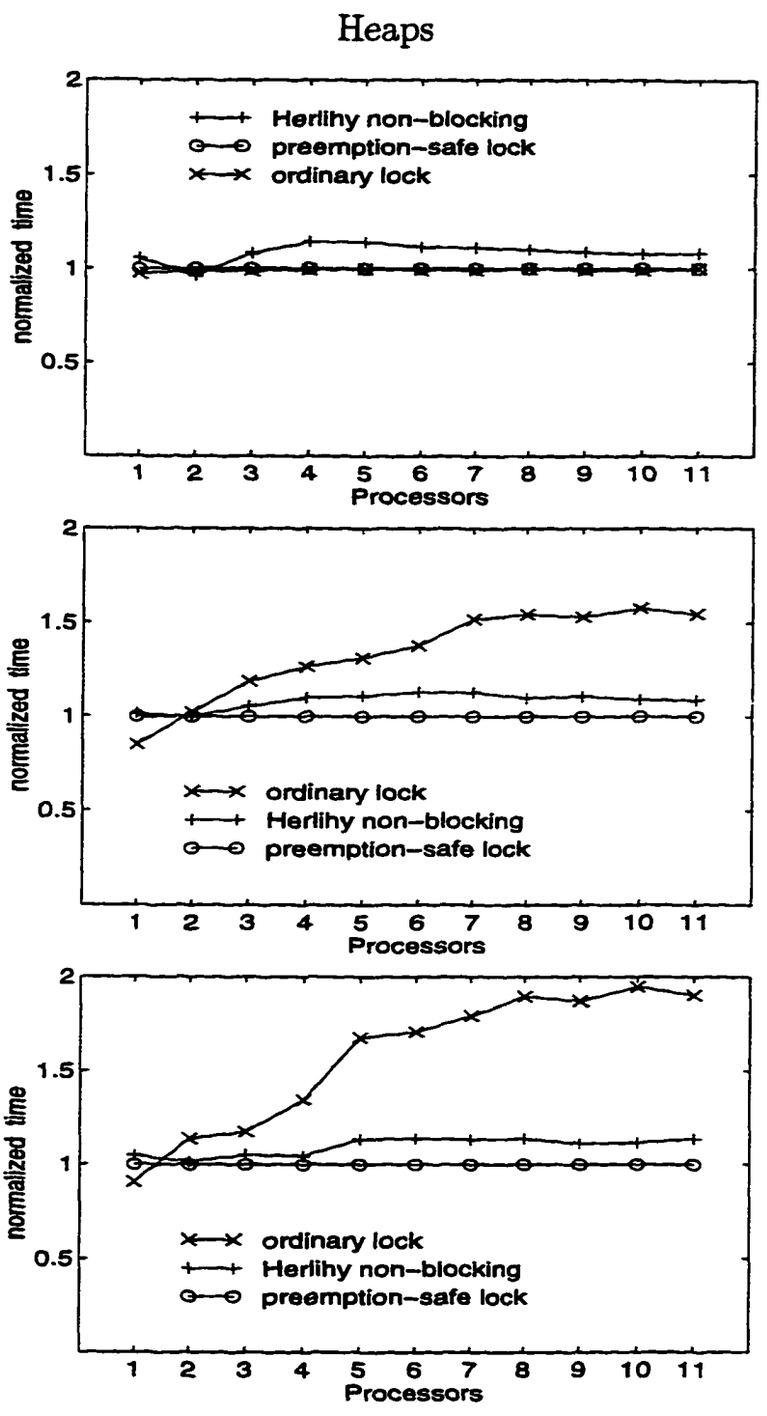

Figure 3.6: Normalized execution time for one million insert/delete_min pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

the copied nodes become part of the global structure and the copied nodes are recycled for use in future operations.

The axes in the graphs have the same semantics as those for the queue and stack graphs. Execution time is normalized to that of the preemption-safe single lock algorithm. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with 1, 2, and 3 processes per processor, are 21.0 and 27.7, 43.1 and 27.4, and 65.0 and 27.6, respectively. Each process executes 1,000,000/p insert/delete_min pairs on an initially empty heap with a 6 μs average delay between successive operations. Experiments with non-empty heaps resulted in relative performance similar to that depicted in the graphs.

As the level of multiprogramming increases the performance of ordinary locks degrades, while the performance of the preemption-safe and non-blocking algorithms remains relatively unchanged. The degradation of the ordinary locks is larger than that suffered by the locks in the queue and stack implementations, because the heap operations are more complex and result in higher levels of contention. Unlike the case for queues and stacks, the non-blocking implementation of heaps is quite complex. It cannot match the performance of the preemption-safe lock implementation on either dedicated or multiprogrammed systems, with or without contention. Heap implementations resulting from general non-blocking methodologies (without data-structure-specific elimination of copying) are even more complex, and could be expected to perform much worse.

COUNTERS	LL/SC	14.6
	Ordinary lock	16.0
	Preemption-safe lock	17.7

Table 3.4: Execution times in seconds for one million atomic increments on a single processor (no contention).

3.4.4 Counters

Figure 3.7 shows performance results for three implementations of counters on a dedicated system, and on multiprogrammed systems with 2 and 3 processes per processor. Table 3.4 shows performance on a dedicated processor. The three implementations are: the usual single-lock algorithm using ordinary and preemption-safe locks, and the non-blocking algorithm using `load_linked/store_conditional`.

The axes in the graphs have the same semantics as those for the previous graphs. Execution time is normalized to that of the preemption-safe single lock algorithm. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with 1, 2, and 3 processes per processor, are 17.7 and 10.8, 35.0 and 11.3, and 50.6 and 10.9, respectively. Each process executes 1,000,000/p increments on a shared counter with a 6 μs average delay between successive operations.

The results are similar to those observed for queues and stacks, but are even more pronounced. The non-blocking algorithm outperforms the preemption-safe lock-based counter by more than 55% on 11 processors with various levels of

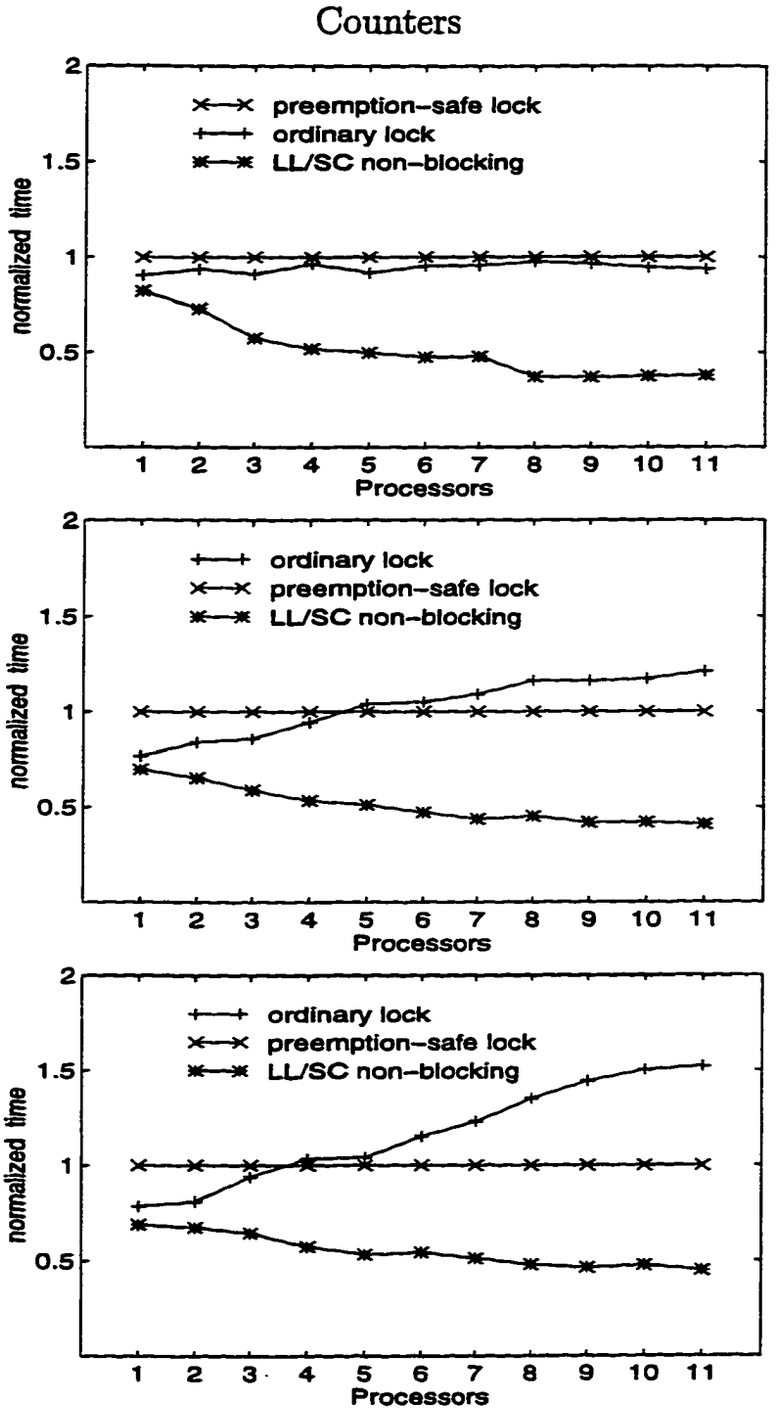

Figure 3.7: Normalized execution time for one million atomic increments on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

QUICKSORT – QUEUE		
	MS non-blocking	3.6
	Single ordinary lock	4.0
	Single safe lock	4.0

Table 3.5: Execution times in seconds for quicksort of 500,000 items using a shared queue on a single processor (no contention).

QUICKSORT – STACK		
	Treiber non-blocking	3.0
	Single ordinary lock	3.3
	Single safe lock	3.4

Table 3.6: Execution times in seconds for quicksort of 500,000 items using a shared stack on a single processor (no contention).

multiprogramming. The performance of a `fetch_and_add` atomic primitive would be even better as we show in Chapter 4.

3.4.5 Quicksort Application

We performed experiments on two versions of a parallel quicksort application, one that uses a link-based queue, and another that uses a link-based stack to distribute items to be sorted among the cooperating processes. We used three implementations for each of the queue and the stack: the usual single lock algorithm using ordinary and preemption-safe locks, and our non-blocking queue and Treiber's stack, respectively. In each execution, the processes cooperate in sorting an array of 500,000 pseudo-random numbers using quicksort for intervals of more than 20 elements, and insertion sort for smaller intervals.

Figure 3.8 and Table 3.5 show performance results for the three queue-based

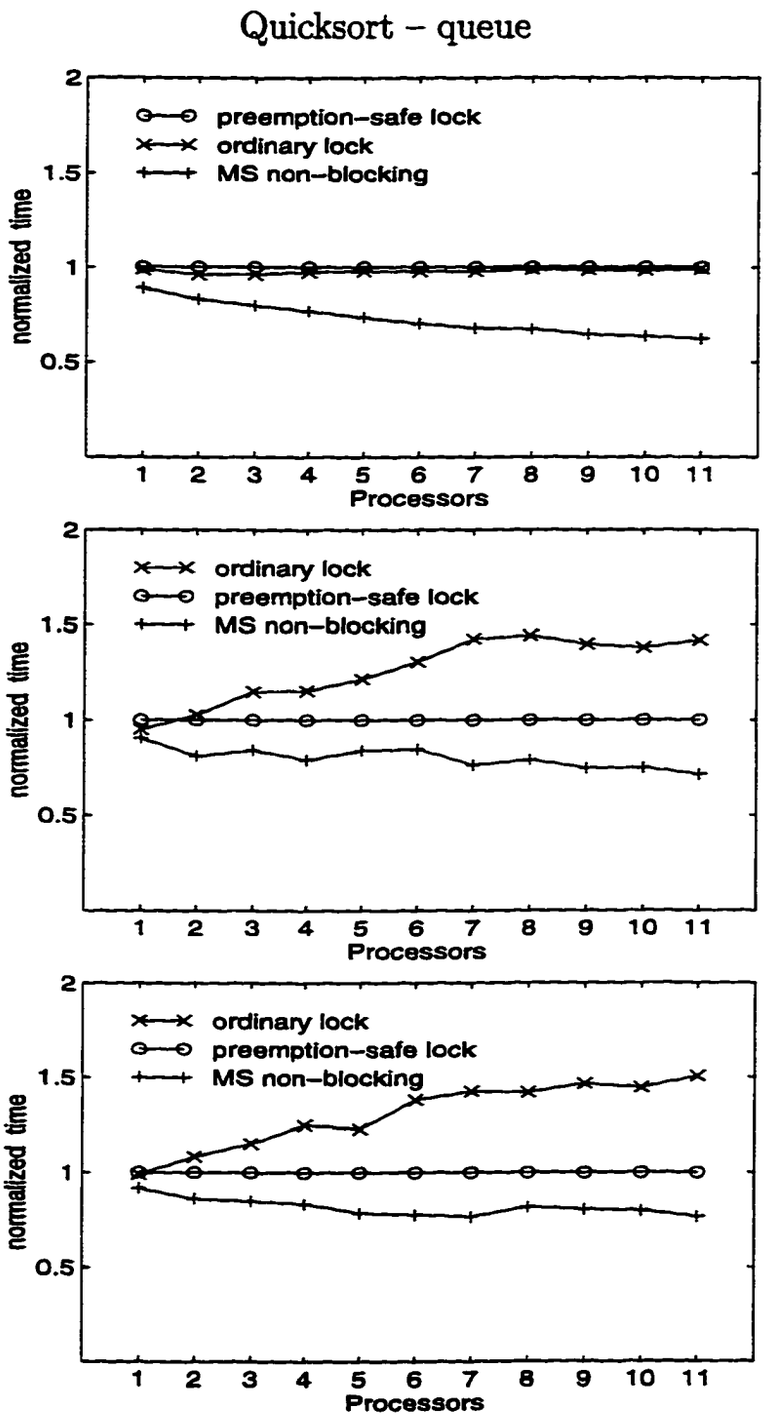

Figure 3.8: Normalized execution time for quicksort of 500,000 items using a shared queue on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

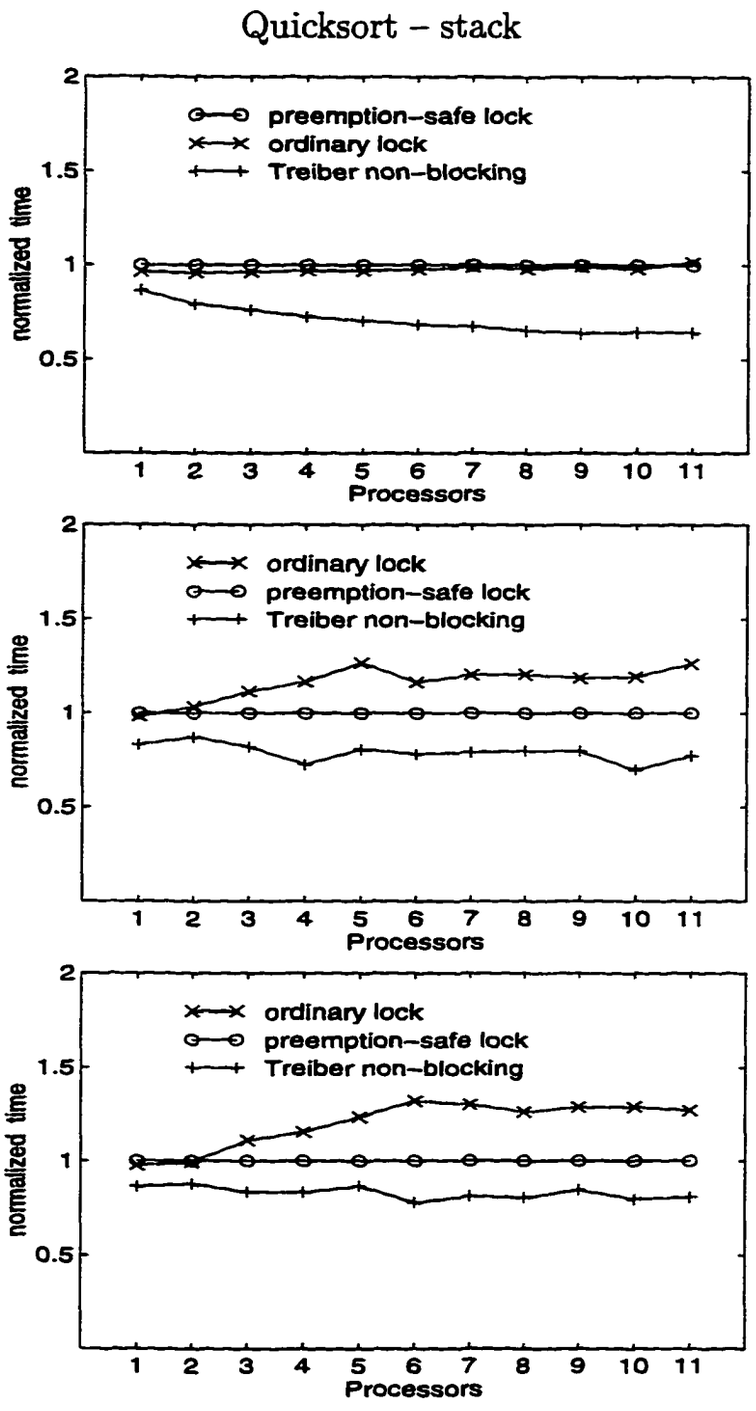

Figure 3.9: Normalized execution time for quicksort of 500,000 items using a shared stack on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

TSP	Ordinary locks	32.8
	Hybrid	33.7
	Non-blocking	34.3
	Safe locks	34.9

Table 3.7: Execution times in seconds for a 17-city traveling salesman problem using a shared priority queue, stack and counters on a single processor (no contention).

versions; Figure 3.9 and Table 3.6 show results for the three stack-based versions. Execution times are normalized to those of the preemption-safe lock-based algorithms. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with 1, 2, and 3 processes per processor, are 4.0 and 1.6, 7.9 and 2.3, and 11.6 and 3.3, respectively for a shared queue, and 3.4 and 1.5, 7.0 and 2.3, and 10.2 and 3.1, respectively for a shared stack.

The results confirm our observations from experiments on micro-benchmarks. Performance with ordinary locks degrades under multiprogramming, though not as severely as before, since more work is being done between atomic operations. Simple non-blocking algorithms yield superior performance even on dedicated systems, making them the strategy of choice under any level of contention or multiprogramming.

3.4.6 Traveling Salesman Application

We performed experiments on a parallel implementation of a solution to the traveling salesman problem. The program uses a shared heap, stack, and counters.

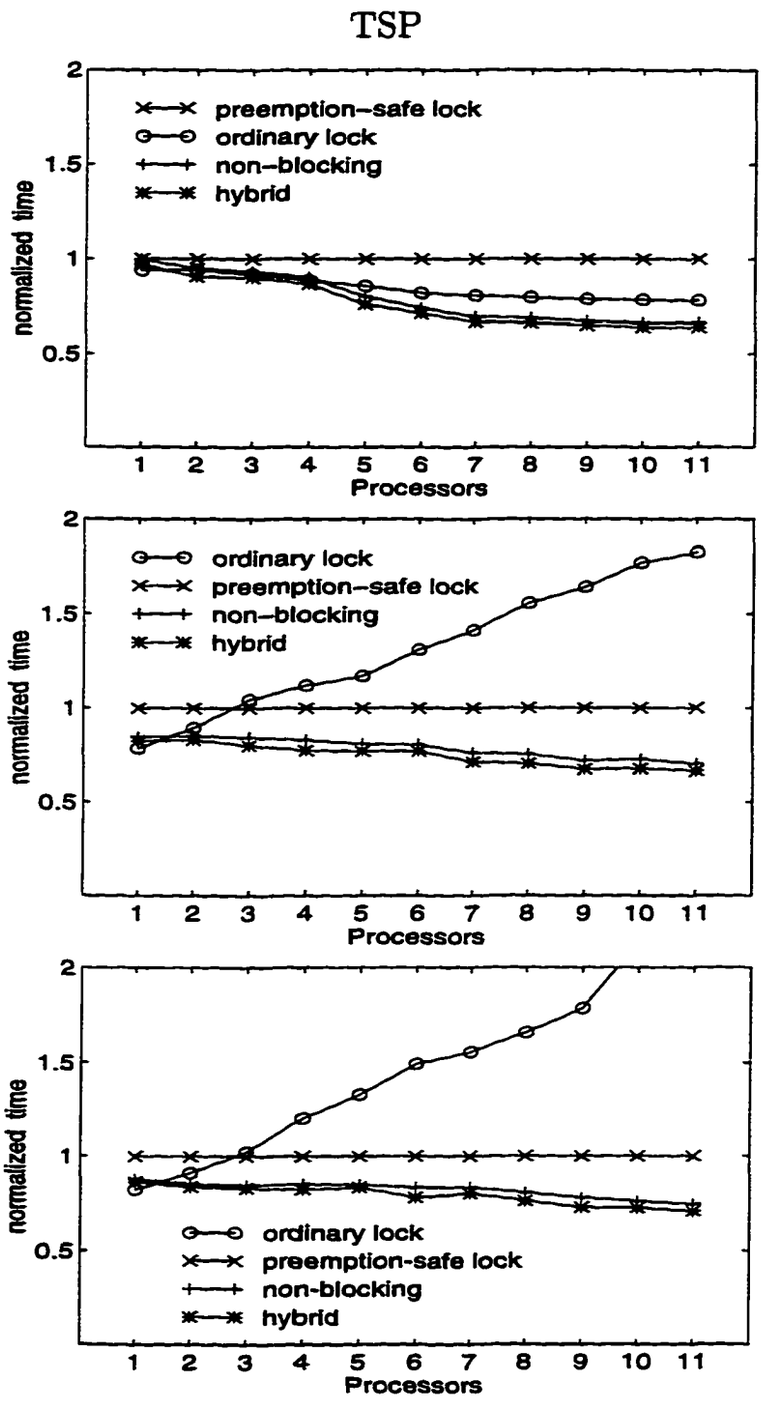

Figure 3.10: Normalized execution time for a 17-city traveling salesman problem using a shared priority queue, stack and counters on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

We used three implementations for each of the heap, stack, and counters: the usual single lock algorithm using ordinary and preemption-safe locks, and the best respective non-blocking algorithms (Herlihy–optimized, Treiber, and `load_linked/store_conditional`). In each execution, the processes cooperate to find the shortest tour in a 17-city graph. The processes use the priority queue heap to share information about the most promising tours, and the stack to keep track of the tours that are yet to be computed. We ran experiments with each of the three implementations of the data structures. In addition, we ran experiments with a "hybrid" program that uses the version of each data structure that ran the fastest for the micro-benchmarks: non-blocking stacks and counters, and a preemption-safe priority queue.

Figure 3.10 and Table 3.7 show performance results for the four different experiments. Execution times are normalized to those of the preemption-safe lock-based experiment. The absolute times in seconds for the preemption-safe lock-based experiment on one and 11 processors, with 1, 2, and 3 processes per processor, are 34.9 and 14.3, 71.7 and 15.7, and 108.0 and 18.5, respectively. Confirming our results with micro-benchmarks, the experiment based on ordinary locks suffers under multiprogramming. The hybrid experiment yields the best performance, since it uses the best implementation of each of the data structures.

3.5 Summary

For atomic update of a shared data structure, the programmer may ensure consistency using (1) a single lock, (2) multiple locks, (3) a general-purpose non-blocking technique, or (4) a special-purpose (data-structure-specific) non-blocking algorithm. The locks in (1) and (2) may or may not be preemption-safe.

Options (1) and (3) are easy to generate, given code for a sequential version of the data structure, but options (2) and (4) must be developed individually for each different data structure. Good data-structure-specific multi-lock and non-blocking algorithms are sufficiently tricky to devise that each has tended to constitute an individual publishable result.

Our experiments indicate that for simple data structures, special-purpose non-blocking atomic update algorithms will outperform all alternatives, not only on multiprogrammed systems, but on dedicated machines as well. Given the availability of a universal atomic hardware primitive, there seems to be no reason to use any other version of a link-based stack, a link-based queue, or a small, fixed-sized object like a counter.

For more complex data structures, however, or for machines without universal atomic primitives, preemption-safe locks are clearly important. Preemption-safe locks impose a modest performance penalty on dedicated systems, but provide dramatic savings on time-sliced systems.

For the designers of future systems, we recommend that kernel interfaces provide a mechanism for preemption-safe locking. For small-scale machines, the Synunix interface [15] appears to work well. For larger machines, a more elaborate interface may be appropriate [41].

Almost all multiprocessors architectures provide support for locks either using special hardware or by supporting atomic primitives such as `test_and_set` or `fetch_and_increment`. On the other hand, non-blocking implementations require architectural support for one of the universal primitives `compare_and_swap` or the pair `load_linked` and `store_conditional`. These primitives are already supported on several multiprocessor architectures [51; 80]. These primitives can provide the functionality of other primitives very efficiently, and some mutual exclusion locks such the queue-based MCS lock require `compare_and_swap` (`load_linked` and `store_conditional` can be used) in order to guarantee FIFO order. Moreover, `load_linked` and `store_conditional` provide a faster implementation of a test-and-`test_and_set` lock than the `test_and_set` primitive itself, since a failed `store_conditional` does not invalidate other caches while an unnecessary `test_and_set` does. Therefore we recommend that multiprocessor designers support these primitives in the instruction set. They are easy to implement on invalidation-based cache-coherent bus-based machines. In the following chapter we investigate their implementation on cache-coherent distributed shared memory machines.

4 Atomic Primitives and Coherence

4.1 Introduction

Several atomic primitives have been proposed and implemented on DSM architectures. Most of them are special-purpose primitives that are designed to support particular synchronization operations. Examples include `test_and_set` with special semantics on the DASH multiprocessor [48], the QOLB primitives on the Wisconsin Multicube [18] and the IEEE Scalable Coherent Interface standard [23], the full/empty bits on the HEP [81], MIT Alewife [1], and Tera machines [4], and the primitives for locking and unlocking cache lines on the Kendall Square KSR1 [40].

While it is possible to implement arbitrary synchronization mechanisms on top of special-purpose locks, greater concurrency, speed, and fault-tolerance may be achieved by using more general-purpose primitives. General-purpose primi-

tives such as `fetch_and_Φ`, `compare_and_swap`, and the pair `load_linked/store_conditional` can easily and efficiently implement a wide variety of styles of synchronization (e.g. operations on wait-free and lock-free objects, read-write locks, priority locks, etc.). These primitives are easy to implement in the snooping protocols of bus-based multiprocessors, but there are many tradeoffs to be considered when developing implementations for a DSM machine.

We propose and evaluate several implementations of these general-purpose atomic primitives on directory-based cache coherent DSM multiprocessors, in an attempt to answer the question: which atomic primitives should be provided on future DSM multiprocessors and how should they be implemented?

Our analysis and experimental results suggest that good overall performance will be achieved by `compare_and_swap`, with comparators in the caches, a write-invalidate coherence policy, and an auxiliary `load_exclusive` instruction.

In section 4.2 we discuss the differences in functionality and expressive power among the primitives under consideration. In section 4.3 we present several implementation options for the primitives under study on DSM multiprocessors. Then we present our experimental results and discuss their implications in section 4.4, and conclude with summary and recommendations in section 4.5.

4.2 Atomic Primitives

4.2.1 Functionality

A `fetch_and_`Φ primitive [19] takes (conceptually) two parameters: the address of the destination operand, and a value parameter. It atomically reads the original value of the destination operand, computes the new value as a function Φ of the original value and the value parameter, stores this new value, and returns the original value. Examples of `fetch_and_`Φ primitives include `test_and_set`, `fetch_and_store`, `fetch_and_add`, and `fetch_and_or`.

The `compare_and_swap` primitive was first provided on the IBM System/370 [13]. `Compare_and_swap` takes three parameters: the address of the destination operand, an expected value, and a new value. If the original value of the destination operand is equal to the expected value, the former is replaced by the new value (atomically) and the return value indicates success, otherwise the return value indicates failure. `Compare_and_swap` is implemented on the Intel x86, the Sparc V9, and the Motorola 68000 architectures.

The pair `load_linked/store_conditional`, proposed by Jensen *et al.* [36], are implemented on the MIPS II [38], the DEC Alpha [80], and the PowerPC [12] architectures. They must be used together to read, modify, and write a shared location. `Load_linked` returns the value stored at the shared location and sets a *reservation* associated with the location and the processor. `Store_conditional`

checks the reservation. If it is valid a new value is written to the location and the operation returns success, otherwise it returns failure. Conceptually, for each shared memory location there is a reservation bit associated with each processor. Reservations for a shared memory location are invalidated when that location is written by any processor. Till recently, `Load_linked` and `store_conditional` have not been implemented on network-based multiprocessors.[1] On bus-based multiprocessors they can easily be embedded in a snooping cache coherence protocol, in such a way that should `store_conditional` fail, it fails locally without causing any bus traffic.

In practice, processors are generally limited to one outstanding reservation, and reservations may be invalidated even if the variable is not written. On the MIPS R4000 [51], for example, reservations are invalidated on context switches and TLB exceptions. We can ignore these spurious invalidations with respect to lock-freedom, so long as we always try again when a `store_conditional` fails, and so long as we never put any instructions between `load_linked` and `store_conditional` that may invalidate reservations deterministically. Depending on the processor, these things may include loads, stores, and incorrectly-predicted

[1]The pair `load_linked/store_conditional` is supprted on the SGI Origin CC-NUMA architecture. They are processed in the processor cache controllers, and cache misses on `load_linked` and `store_conditional` are handled by the NUMA coherence protocol as regular loads and stores, respectively. Our work presented in this chapter predates the production of the Origin multiprocessors. The NUMA protocol associated with the in-cache implementation presented in this chapter differs from that of the Origin in that it conytains a potential optimization that it aborts transfering the data to the requesting node for a `store_conditional` cache miss as soon as it determines that the `store_conditional` is doomed to fail.

branches.

4.2.2 Expressive Power

Herlihy introduced an impossibility and universality hierarchy [26] that ranks atomic operations according to their relative power. The hierarchy is based on the concepts of lock-freedom and wait-freedom. A concurrent object implementation is *lock-free* if it always guarantees that some processor will complete an operation in a finite number of steps, and it is *wait-free* if it guarantees that each process will complete an operation in a finite number of steps. Lock-based operations are neither lock-free nor wait-free. In Herlihy's hierarchy, it is impossible for an atomic operation at a lower level of the hierarchy to provide a lock-free implementation of an atomic operation in a higher level. Atomic load and store are at level 1. The primitives `fetch_and_store`, `fetch_and_add`, and `test_and_set` are at level 2. `Compare_and_swap` is a universal primitive—it is at level ∞ of the hierarchy [29]. `Load_linked/store_conditional` can also be shown to be universal if we assume that reservations are invalidated *if and only if* the corresponding shared location is written to.

Thus, according to Herlihy's hierarchy, `compare_and_swap` and `load_linked/store_conditional` can provide lock-free simulations of `fetch_and_`Φ primitives, and it is impossible for a `fetch_and_`Φ primitive to provide a lock-free simulation of `compare_and_swap` or `load_linked/store_conditional`. It should also be noted

that although `fetch_and_store` and `fetch_and_add` are at the same level (level 2) in Herlihy's hierarchy, this does not imply that there are lock-free simulations of one of these primitives using the other. Similarly, while both `compare_and_swap` and the pair `load_linked/store_conditional` are universal primitives, it is possible to provide a simple lock-free simulation of `compare_and_swap` using `load_linked` and `store_conditional`, but not vice versa.

A pair of `atomic_load` and `compare_and_swap` cannot simulate `load_linked` and `store_conditional` because of the ABA problem, discussed in Section 2.2. Herlihy presented methodologies for implementing lock-free (and wait-free) implementations of concurrent data objects using `compare_and_swap` [27] and `load_linked/store_conditional` [30]. The `compare_and_swap` algorithms are less efficient and conceptually more complex than the `load_linked/store_conditional` algorithms due to the pointer problem [30].

On the other hand, there are several algorithms that need or benefit from `compare_and_swap` [41; 54; 56; 57]. A simulation of `compare_and_swap` using `load_linked` and `store_conditional` is less efficient than providing `compare_and_swap` in hardware. A successful simulated `compare_and_swap` is likely to cause two cache misses instead of the one that would occur if `compare_and_swap` were supported in hardware. (If `load_linked` suffers a cache miss, it will generally obtain a shared (read-only) copy of the line. `Store_conditional` will miss again in order to obtain write permission.) Also, unlike `load_linked/store_conditional`, `compare_and_`

swap is not subject to any restrictions on the loads and stores between atomic_load and compare_and_swap. Thus, it is more suitable for implementing atomic update operations that require memory access between loading and comparing (e.g. an atomic update operation that requires a table lookup based on the original value).

4.3 Implementations

The main design issues for implementing atomic primitives on cache coherent DSM multiprocessors are:

1. Where should the computational power to execute the atomic primitives be located: in the cache controllers, in the memory modules, or both?
2. Which coherence policy should be used for atomically accessed data: no caching, write-invalidate, or write-update?
3. What auxiliary instructions, if any, can be used to enhance performance?

We focus our attention on fetch_and_Φ, compare_and_swap, and load_linked/store_conditional because of their generality, their popularity on small-scale machines, and their prevalence in the literature. We consider three implementations for fetch_and_Φ, five for compare_and_swap, and three for load_linked/store_conditional. The implementations can be grouped into three categories according to the coherence policies used:

1. INV (INValidate): Computational power in the cache controllers, with a write-invalidate coherence policy. The main advantage of this implementation is that once the datum is in the cache, subsequent atomic updates are executed locally, so long as accesses by other processors do not intervene.

2. UPD (UPDate): Computational power in the memory, with a write-update policy. The main advantage of this implementation is a high read hit rate, even in the case of alternating accesses by different processors.

3. UNC (UNCached): Computational power in the memory, with caching disabled. The main advantage of this implementation is that it eliminates the coherence overhead of the other two policies, which may be a win in the case of high contention or even the case of no contention when accesses by different processors alternate.

INV and UPD implementations are embedded in the cache coherence protocols. Our protocols are mainly based on the directory-based protocol of the DASH multiprocessor [48].

For `fetch_and_`Φ and `compare_and_swap`, INV obtains an exclusive copy of the datum and performs the operation locally. UNC sends a request to the memory to perform the operation on an uncached datum. UPD also sends a request to the memory to perform the operation, but retains a shared copy of the datum in the local cache. The memory sends updates to all the caches with copies.

In addition, for `compare_and_swap` we consider two variants of INV: INVd ('d' for deny) and INVs ('s' for share). In these variants, if the line is not already cached in exclusive mode locally, comparison of the old value with the expected value takes place in either the home node or the owner node, whichever has the most up-to-date copy of the line (the home node is the node at which the memory resides; the owner, if any, is the node that has an exclusive cached copy of the line). If equality holds, INVd and INVs behave like INV: the requesting node acquires an exclusive copy. Otherwise, the response to the requesting node indicates that `compare_and_swap` must fail. In the case of INVd, no cached copy is provided. In the case of INVs, a read-only copy is provided. The rationale behind these variants is to prevent a request that will fail from invalidating copies cached in other nodes.

The implementations of `load_linked/store_conditional` are somewhat more elaborate, due to the need for reservations. In the INV implementation, each processing node has a reservation bit and a reservation address register. `Load_linked` sets the reservation bit to valid and writes the address of the shared location to the reservation register. If the cache line is not valid, a shared copy is acquired, and the value is returned. If the cache line is invalidated and the address corresponds to the one stored in the reservation register, the reservation bit is set to invalid. `Store_conditional` checks the reservation bit. If it is invalid, `store_conditional` fails. If the reservation bit is valid and the line is exclusive, `store_`

`conditional` succeeds locally. Otherwise, the request is sent to the home node. If the directory indicates that the line is exclusive or uncached, `store_conditional` fails, otherwise (the line is shared) `store_conditional` succeeds and invalidations are sent to holders of other copies.

In the UNC implementation of `load_linked`/ `store_conditional`, each memory location (at least conceptually) has a reservation bit vector of size equal to the total number of processors. `Load_linked` reads the value from memory and sets the appropriate reservation bit. Any write or successful `store_conditional` to the location clears the reservation vector. `Store_conditional` checks the corresponding reservation bit and succeeds or fails accordingly. Various space optimizations are conceivable for practical implementations; see section 4.3.1 below. In the UPD implementation, `load_linked` requests have to go to memory even if the datum is cached, in order to set the appropriate reservation bit. Similarly, `store_conditional` requests have to go to memory to check the reservation bit.

We consider two auxiliary instructions. `Load_exclusive` reads a datum but acquires exclusive access. It can be used with INV instead of an ordinary load when reading a datum that is then accessed by `compare_and_swap`. The intent is to make it more likely that `compare_and_swap` will not have to go to memory. `Load_exclusive` is also useful for ordinary operations on migratory data. `Drop_copy` can be used to drop (self-invalidate) a cached line, to reduce the number of serialized messages required for subsequent accesses by other processors. A

write miss on an uncached datum requires 2 serialized messages (from requesting node to the home node and back), instead of 4 for a remote exclusive datum (requesting node to home to owner to home and back to requesting node) and 3 for a remote shared datum (from requesting node to home to sharing nodes, with acknowledgments sent back to the requesting node).

4.3.1 Hardware Requirements

If the base coherence policy is different from the coherence policy for access to synchronization variables, the complexity of the cache coherence protocol increases significantly. However, the directory entry size remains the same with any coherence policy on directory-based multiprocessors (modulo any requirements for reservation information in the memory for `load_linked/store_conditional`).

Computational power (e.g. adders or comparators) needs to be added to each cache controller if the implementation is INV, or to each memory module if the implementation is UPD or UNC, or to both caches and memory modules if the implementation for **compare_and_swap** is INVd or INVs.

If `load_linked` and `store_conditional` are implemented in the caches, one reservation bit and one reservation address register per cache are needed to maintain ideal semantics, assuming that `load_linked` and `store_conditional` pairs are not allowed to nest. On the MIPS R4000 processor [51] there is an `LLbit` and an on-chip system control processor register `LLAddr`. The `LLAddr` register

is used only for diagnostic purposes, and serves no function during normal operation. Thus, invalidation of any cache line causes the `LLbit` to be reset. A `store_conditional` to a valid cache line is not guaranteed to succeed, as the datum might have been written by another process on the same physical processor. Thus, a reservation bit is needed (at least to be invalidated on a context switch).

If `load_linked` and `store_conditional` are implemented in the memory, the hardware requirements are more significant. A reservation bit for each processor is needed for each memory location. There are several options:

- A bit vector of size equal to the number of processors can be added to each directory entry. This option limits the scalability of the multiprocessor, as the (total) directory size increases quadratically with the number of processors.

- A linked list can be used to hold the ids of the processors holding reservations on a memory block. The size overhead is reduced to the size of the head of the list, if the memory block has no reservations associated with it. However, a free list is needed and it has to be maintained by the cache coherence protocol.

- A limited number of reservations (e.g. 4) can be maintained. Reservations beyond the limit will be ignored, so their corresponding `store_conditional`'s are doomed to fail. If a failure indicator can be returned by beyond-the-limit

load_linked's, then the corresponding store_conditional's can fail locally without causing any network traffic. This option eliminates the need for bit vectors or a free list. Also, it can help reduce the effect of high contention on performance. However, it compromises the lock-free semantics of lock-free objects based on load_linked and store_conditional.

- A hardware counter associated with each memory block can be used to indicate a serial number of writes to that block. Load_linked will return both the datum and the serial number, and store_conditional must provide both the datum and the expected serial number. A store_conditional with a serial number different from that of the counter will fail. The counter should be large enough (e.g. 32 bits) to eliminate any problems due to wrap-around. The message sizes associated with load_linked and store_conditional must increase by the counter size.

In each of these options, if the space overhead is too high to accept for all of memory, atomic operations can, with some loss of convenience, be limited to a subset of the physical address space.

For the purposes of this study we do not need to fix an implementation for reservations, but we prefer the last one. It has the potential to provide the advantages of both compare_and_swap and load_linked/store_conditional. Load_linked resembles a load that returns a longer datum; store_conditional resembles a compare_and_swap that provides a longer datum. The serial number

portion of the datum eliminates the pointer problem mentioned in section 4.2.2. In addition, the lack of an explicit reservation means that `store_conditional` does not have to be preceded closely in time by `load_linked`; a process that expects a particular value (and serial number) in memory can issue a bare `store_conditional`, just as it can issue a bare `compare_and_swap`. This capability is useful for algorithms such as the MCS queue-based spin lock [56], in which it reduces by one the number of memory accesses required to relinquish the lock. It is not even necessary that the serial number reside in special memory: `load_linked` and `store_conditional` could be designed to work on doubles. The catch is that "ordinary" stores to synchronization variables would need to update the serial number. If this number were simply kept in half of a double, special instructions would need to be used instead of ordinary stores.

4.4 Experimental Results

4.4.1 Methodology

The experimental results were collected using an execution driven cycle-by-cycle simulator. The simulator uses MINT (Mips INTerpreter) [92], which simulates MIPS R4000 object code, as a front end. The back end simulates a 64-node multiprocessor with directory-based caches, 32-byte blocks, queued memory, and a 2-D worm-hole mesh network. The simulator supports directory-based cache co-

herence protocols with write-invalidate and write-update coherence policies. The base cache coherence protocol—used for all data not accessed by atomic primitives in all experiments—is a write-invalidate protocol. In order to provide accurate simulations of programs with race conditions, the simulator keeps track of the values of cached copies of atomically accessed data in the cache of each processing node. In addition to the MIPS R4000 instruction set (which includes `load_linked` and `store_conditional`), the simulated multiprocessor supports `fetch_and_`Φ, `compare_and_swap`, `load_exclusive`, and `drop_copy`. Memory and network latencies reflect the effect of memory contention and of contention at the entry and exit of the network (though not at internal nodes).

We used two sets of applications, real and synthetic, to achieve different goals. We began by studying two lock-based applications from the SPLASH suite [78]—LocusRoute and Cholesky—and an application that computes the transitive closure of a directed graph—based on the Floyd-Warshall algorithm [14]—that uses a lock-free counter to distribute variable-size input-dependent jobs among the processors (Figure 4.1). From these real applications we identified typical sharing patterns of atomically accessed data (see Section 4.4.2. In LocusRoute and Cholesky, we replaced the library locks with an assembly language implementation of the test-and-`test_and_set` lock [75] with bounded exponential backoff implemented using the atomic primitives and auxiliary instructions under study. In Transitive Closure, we used different versions of a lock-free counter using `fetch_`

```
private pid, procs, size;
shared  counter, flag, E;

tclosure()
{
local  i,j,k,row,rows,work,pivot,cur;

  for(i=0;i<size;i++)
  {
    if(pid==0){ counter=0; flag=0; }
    row=0;  rows=0;
    barrier();
    while(!flag)
    {
      rows=((size-row-rows-1)>>1)/procs+1;
      row=fetch_and_add(&count,rows);
      if(row>=size){ flag=1; break;}
      work=(rows<size-row) ? rows : size-row;
      pivot=E[i];
      for(j=row;j$<$row+work;j++)
      {
        cur=E[j];
        if((cur[i]==TRUE) && (i!=j))
          for(k=0;k<size;k++)
            if(pivot[k]==TRUE) cur[k]=TRUE;
      }
    }
    barrier();
  }
}
```

Figure 4.1: Transitive closure program for process pid.

and_add, compare_and_swap, and load_linked/store_conditional, and with a scalable tree barrier [56] for barrier synchronization.

Our three synthetic applications served to explore the parameter space and to provide controlled performance measurements. The first uses a lock-free con-

current counter to cover the case in which `load_linked/store_conditional` and `compare_and_swap` simulate `fetch_and_`Φ (specifically `fetch_and_add`). The second uses a counter protected by a test-and-`test_and_set` lock with bounded exponential backoff to cover the case in which all three primitives (`load_linked/store_conditional`, `compare_and_swap` and `fetch_and_`Φ) are used in a similar manner (i.e. spinning). The third uses a counter protected by an MCS lock [56] to cover the case in which `load_linked/store_conditional` simulates `compare_and_swap`.

4.4.2 Sharing Patterns

Performance of atomic primitives is affected by contention and average write-run length [16]. We define the level of contention to be the number of processors that concurrently try to access an atomically accessed shared location. Average write-run length is the average number of consecutive writes (including atomic updates) by a processor to an atomically accessed shared location without intervening accesses (reads or writes) by any other processors.

The average write-run length of atomically accessed data in simulated runs of LocusRoute and Cholesky on 64 processors with different coherence policies was found to range from 1.70 to 1.83, and from 1.59 to 1.62, respectively. This indicates that in these applications lock variables are unlikely to be written more than two consecutive times by the same processor without intervening accesses by other processors. In other words, a processor usually acquires and releases a lock

without intervening accesses by other processors, but it is unlikely to re-acquire it without intervention. In Transitive Closure, the average write-run length was found to be always slightly above 1.00, suggesting a very high level of contention as shown in the next paragraph.

As a measure of contention, we plot the number of processors contending to access a shared location at the beginning of each access (we found a line graph to be more readable than a bar graph, though the results are discrete, not continuous). Figure 4.2 shows the contention histograms for the real applications, with different coherence policies. The figures for LocusRoute and Cholesky indicate that the no-contention case is the common one, for which performance should be optimized. At the same time, they indicate that the low and moderate contention cases do arise, so that performance for them needs also to be good. High contention is rare: reasonable differences in performance among the primitives can be tolerated in this case. However, the figure for Transitive Closure—which achieves an acceptable efficiency of 45% on 64 processors—indicates a common case of very high contention, implying that differences in performance among the primitives are more important in this case. The contention can be attributed to the frequent use of barrier synchronization in the application, which increases the likelihood that all or most of the processors will try to access the counter concurrently.

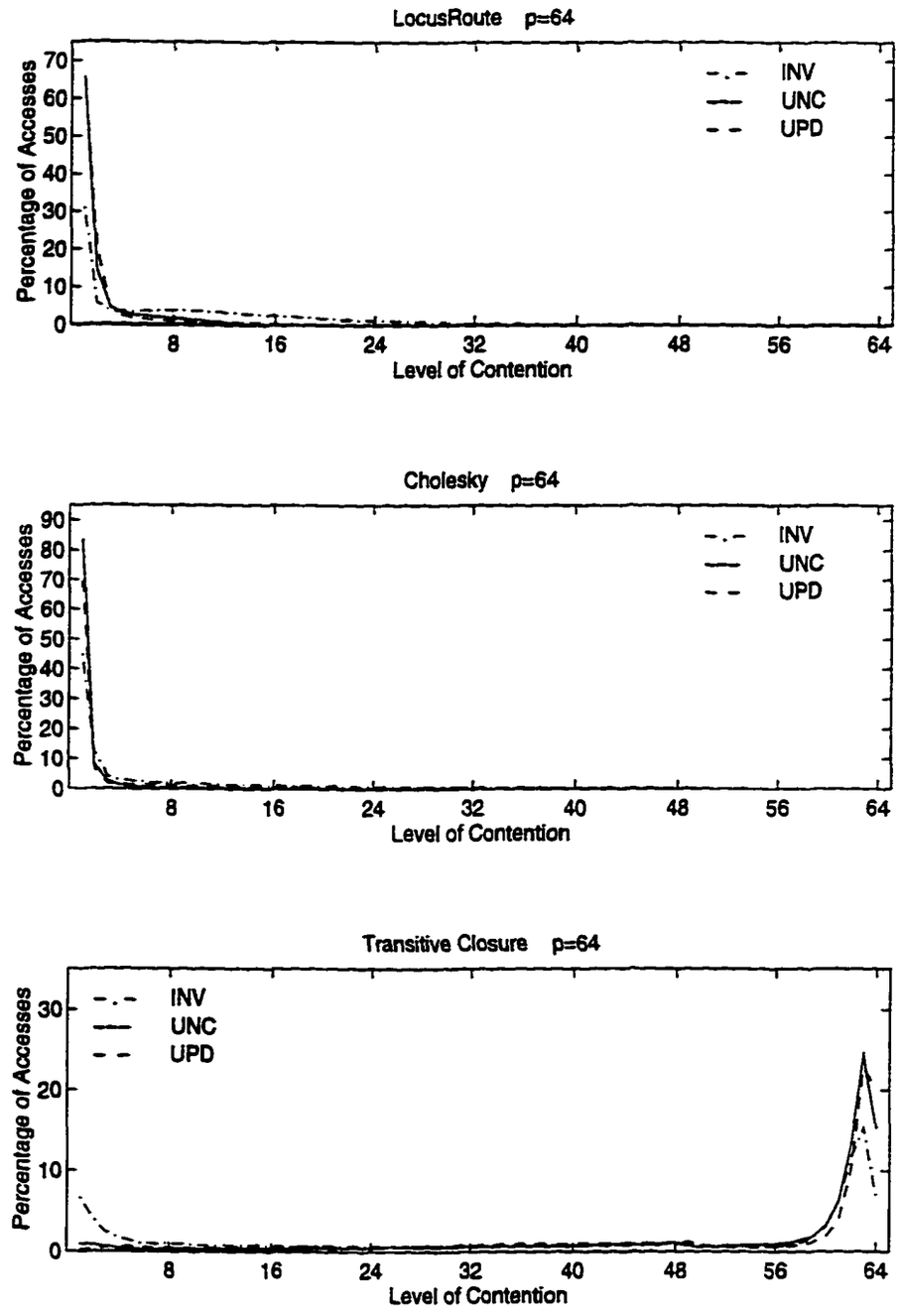

Figure 4.2: Histograms of the level of contention in LocusRoute, Cholesky, and Transitive Closure.

4.4.3 Relative Performance of Implementations

We collected performance results of the synthetic applications with various levels of contention and write-run length. We used artificial constant-time barriers supported by MINT to control the level of contention. Because these barriers are constant-time, they have no effect on the results other than enforcing the intended sharing patterns. In these applications, each processor executes a tight loop, in each iteration of which it either updates the counter or not, depending on the desired level of contention. Depending on the desired average write-run length, every one or more iterations are separated by a constant-time barrier.

Figures 4.3, 4.4, and 4.5 show the performance results for the synthetic applications. The bars represent the elapsed time averaged over a large number of counter updates. In each figure, the graphs to the left represent the no-contention case with different numbers of consecutive accesses by each processor without intervention from the other processors. The graphs to the right represent different levels of contention. The bars in each graph are categorized according to the three coherence policies used in the implementation of atomic primitives. In INV and UPD, there are two subsets of bars. The bars to the right represent the results with the `drop_copy` instruction; those to the left represent the results without it. In each of the two subsets in the INV category, there are 4 bars for `compare_and_swap`. These represent, from left to right, the results for the INV, INVd, INVs, and INV with `load_exclusive` implementations.

Figure 4.6 shows the performance results for LocusRoute, Cholesky, and Transitive Closure. Time is measured from the beginning to the end of execution of the parallel part of the applications. The order of bars in the graph is the same as in the previous figures.

We base our analysis on the results of the synthetic applications, where we have control over the parameter space. The results for the real applications serve only to validate the results of the synthetic applications. LocusRoute and Transitive Closure use dynamic scheduling, which explains the difference in relative performance between primitives in these applications and in the corresponding synthetic applications. With dynamic scheduling slight changes in timings allow processors to obtain work from the central work pool in different orders, causing changes in control flow and load balancing.

Coherence Policy

In the case of no contention with short write runs, UNC implementations of the three primitives are competitive with, and sometimes better than, the corresponding cached implementations, even with an average write-run length as large as 2. There are two reasons for these results. First, a write miss on an uncached line takes two serialized messages, which is always the case with UNC, while a write miss on a remote exclusive or remote shared line takes 4 or 3 serialized messages respectively (see table 4.1). Second, UNC implementations do not incur

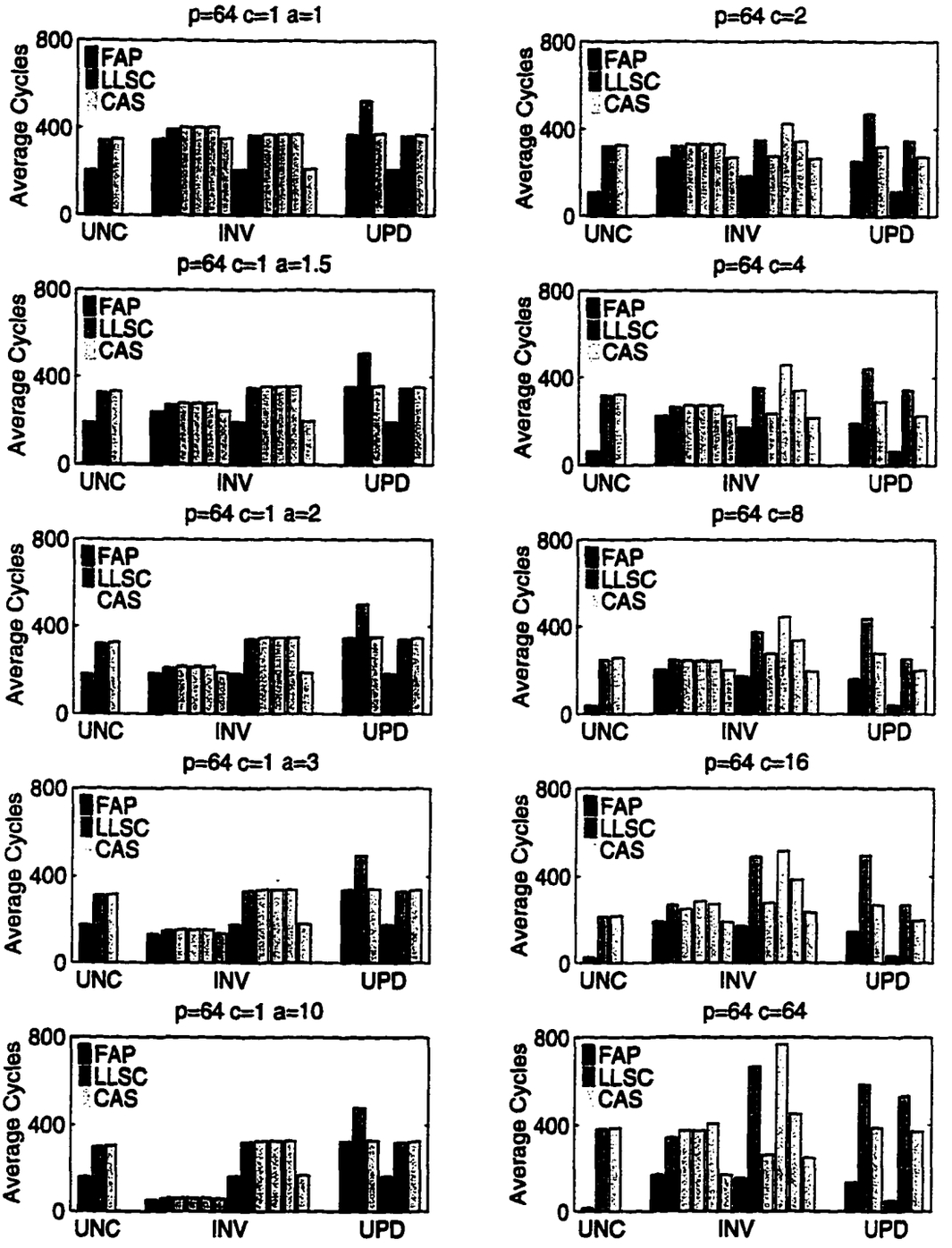

Figure 4.3: Average time per counter update for the lock-free counter application. P denotes processors, c contention, and a the average number of non-interleaved counter updates by each processor.

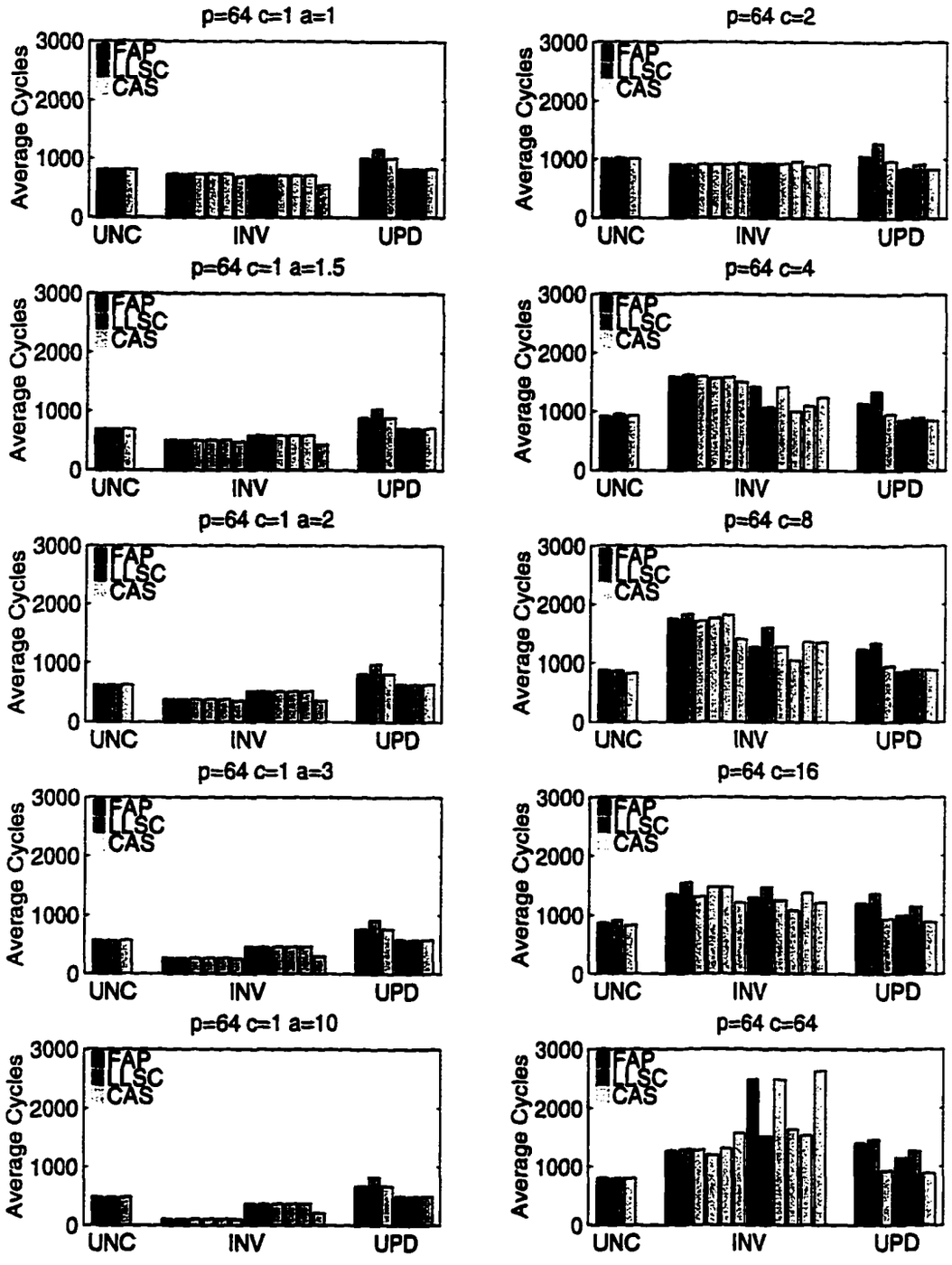

Figure 4.4: Average time per counter update for the TTS-lock-based counter application. P denotes processors, c contention, and a the average number of non-interleaved counter updates by each processor.

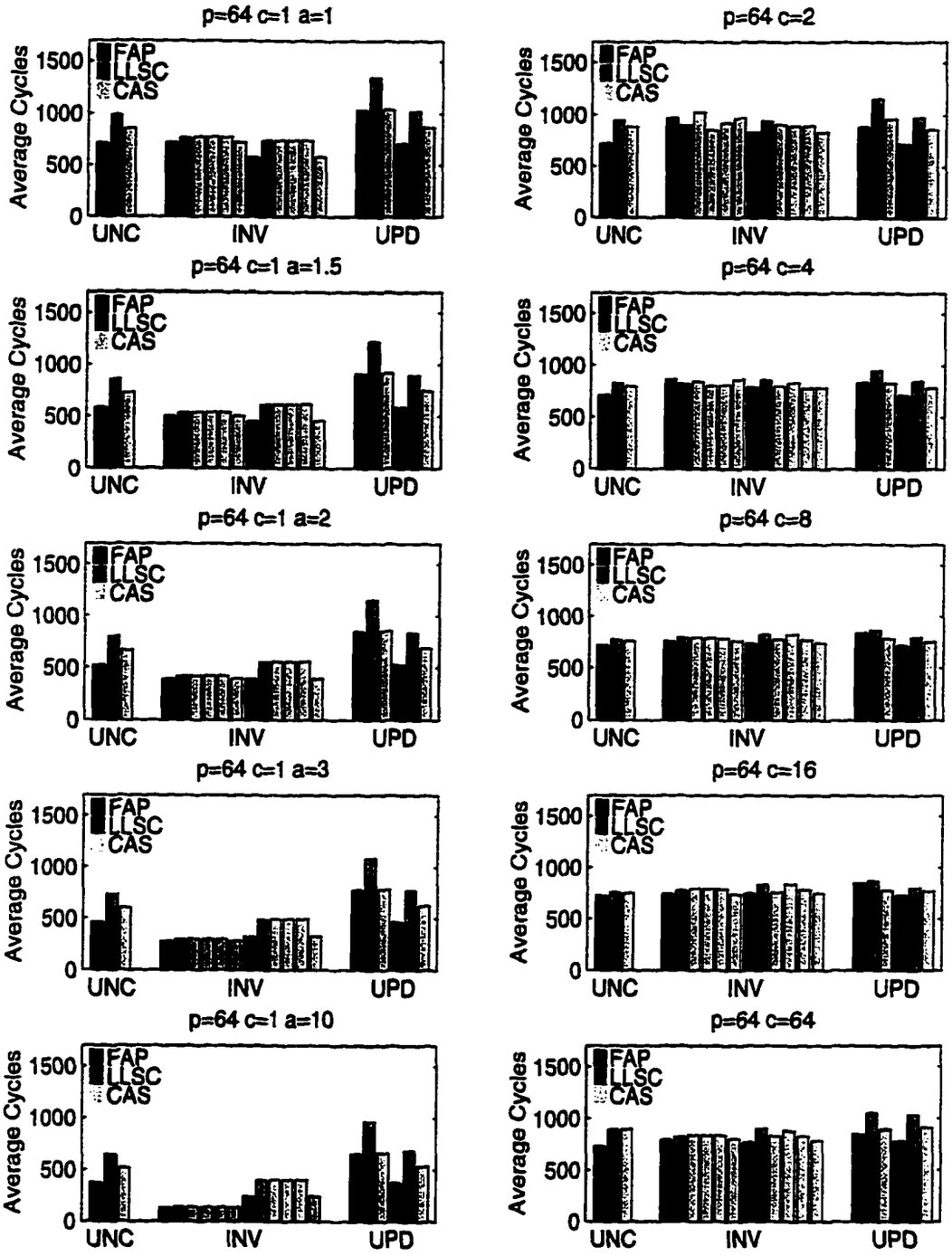

Figure 4.5: Average time per counter update for the MCS-lock-based counter application. P denotes processors, c contention, and a the average number of non-interleaved counter updates by each processor.

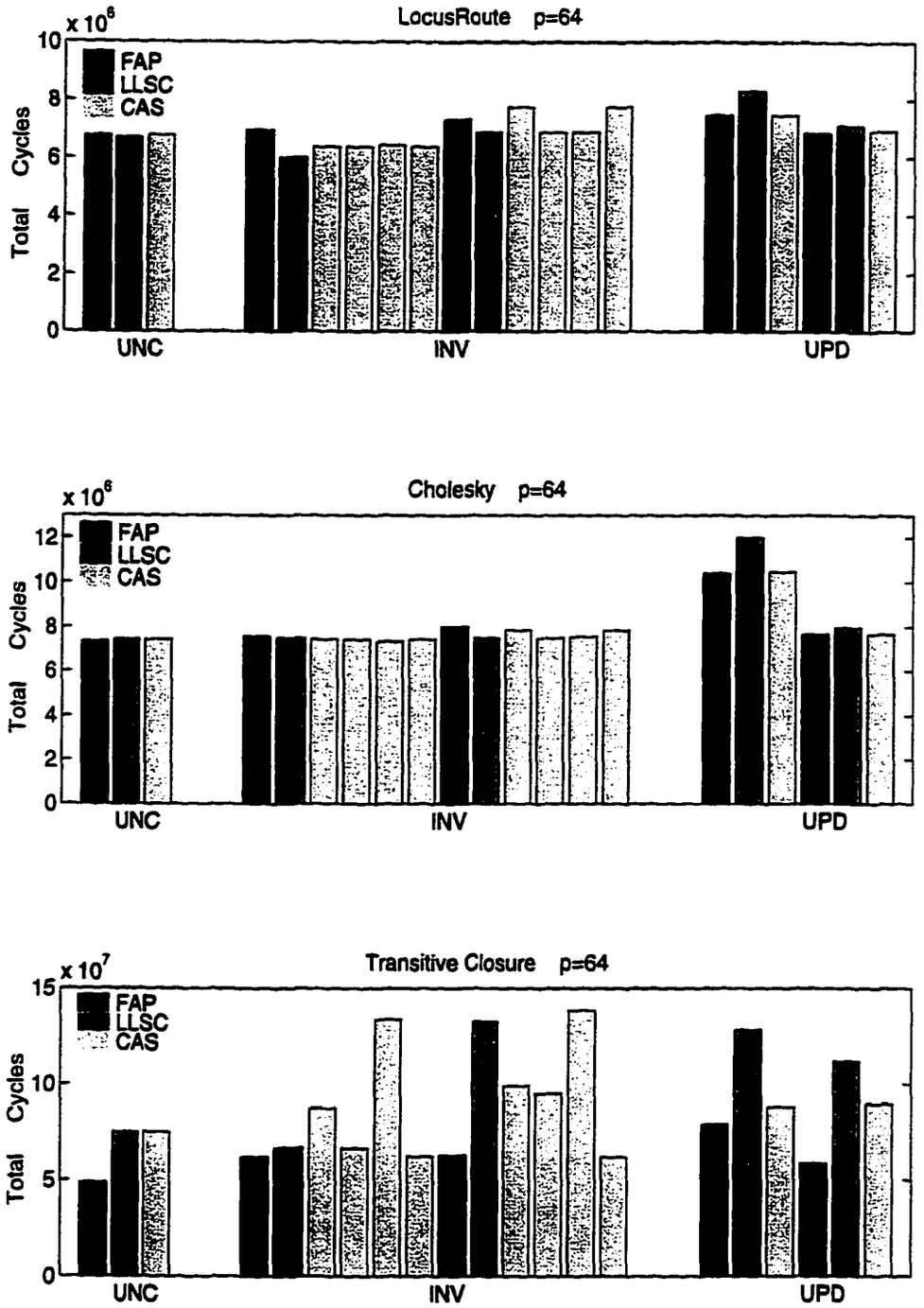

Figure 4.6: Total elapsed time for LocusRoute, Cholesky, and Transitive Closure with different implementations of atomic primitives.

UNC	2
INV to cached exclusive	0
INV to remote exclusive	4
INV to remote shared	3
INV to uncached	2
UPD to cached	3
UPD to uncached	2

Table 4.1: Serialized network messages for stores to shared memory with different coherence policies.

the overhead of invalidations and updates as INV and UPD implementations do.

Furthermore, with contention (even very low), UNC outperforms the other policies (with the exception of INV `compare_and_swap/load_exclusive` when simulating `fetch_and_`Φ), as the effect of avoiding excess serialized messages, and invalidations or updates, is more evident as ownership of data changes hands more frequently. The INV `compare_and_swap/load_exclusive` combination for simulating `fetch_and_`Φ is an exception as the timing window between the read and the write in the read-modify-write cycle is narrowed substantially, thereby diminishing the effect of contention by other processors. Also, in the INV implementation, successful `compare_and_swap`'s after `load_exclusive`'s are mostly hits, while by definition, all UNC accesses are misses.

On the other hand, as write-run length increases, INV increasingly outperforms UNC and UPD, because subsequent accesses in a run are all hits.

Comparing UPD to INV, we find that INV is better in most cases. This is due to the excessive number of useless updates incurred by UPD. INV is much

better in the case of long write runs, as it benefits from caching. In the case of high contention with the test-and-`test_and_set` lock, UPD is better, since every time the lock is released almost all processors try to acquire it by writing to it. With INV all these processors acquire exclusive copies although only one will eventually succeed in acquiring the lock, while in the case of UPD, only successful writes cause updates. However this is not the common case with locks, in which backoff serves to greatly reduce contention.

Atomic Primitives

In the case of the lock-free counter, UNC `fetch_and_add` yields superior performance over the other primitives and implementations, especially with contention. The exception is the case of long write runs, which are not the common case, and may well represent bad programs (e.g. a shared counter should be updated only when necessary, instead of being repeatedly incremented). We conclude that UNC `fetch_and_add` is a useful primitive to provide for supporting shared counters. The `fetch_and_clear_then_add` primitive supported on the BBN Butterfly provides `fetch_and_add`'s capability and the ability to write a datum and update an adjacent counter. However, since `fetch_and_clear_then_add` is not universal, we recommend implementing it only in addition to a universal primitive.

Among the INV universal primitives, `compare_and_swap` almost always benefits from `load_exclusive`, because `compare_and_swap`'s are hits in the case of

no contention and, as mentioned earlier, `load_exclusive` helps minimize the failure rate of `compare_and_swap` as contention increases. In contrast, `load_linked` cannot be exclusive: otherwise livelock is likely to occur.

The performance of the INVd and INVs implementations of `compare_and_swap` is almost always equal to or worse than that of `compare_and_swap` or `compare_and_swap/load_exclusive`. The cost of extra hardware to make comparisons both in memory and in the caches does not appear to be warranted.

As for UPD universal primitives, `compare_and_swap` is always better than `load_linked/store_conditional`, as most of the time `compare_and_swap` is preceded by an ordinary read which is most likely to be a hit with UPD. `Load_linked` requests have to go to memory even if the datum is cached locally, as the reservation has to be set in a unique place that has the most up-to-date version of data—in memory in the case of UPD.

With an INV policy and an average write-run length of one with no contention, `drop_copy` improves the performance of `fetch_and_`Φ and `compare_and_swap/load_exclusive`, because it allows the atomic primitive to obtain the needed exclusive copy of the data with only 2 serialized messages instead of 4 (no other processor has the datum cached; they all have dropped their copies). As contention increases, the effect of `drop_copy` varies with the application. It can in fact cause an *increase* in serialized messages and memory and network contention. For example, an exclusive cache line may be dropped just when its owner is about

to receive a remote request for an exclusive copy of the line. The write-back causes unnecessary memory and network traffic. Moreover, instead of granting the remote request, the local node replies with a negative acknowledgment, and the remote node has to repeat its request for exclusive access to the subsequent owner.

With an UPD policy, `drop_copy` always improves performance, because it reduces the number of useless updates and in most cases reduces the number of serialized messages for a write from 3 to 2.

4.5 Summary

Based on the experimental results and the relative power of atomic primitives, we recommend implementing `compare_and_swap` in the cache controllers of future DSM multiprocessors, with a write-invalidate coherence policy for atomically-accessed data. We also recommend supporting `load_exclusive` to enhance the performance of `compare_and_swap` (as well as assisting in data migration). To address the pointer problem, we recommend consideration of an implementation based on serial numbers, as described for the in-memory implementation of `load_linked/store_conditional` in section 4.3.1.

Although we do not recommend it as the sole atomic primitive, because it is not universal, we find `fetch_and_add` to be very efficient for lock-free counters,

and for many other objects [20]. We recommend implementing it in uncached memory as an extra atomic primitive.

In this chapter we assume a fixed architecture for the coherence controllers in the nodes of the DSM system. In the next chapter, we study the performance of several coherence controller architectures.

5 Coherence Controller Architectures

5.1 Introduction

Previous research has shown that scalable shared-memory performance can be achieved on directory-based cache-coherent multiprocessors such as the Stanford DASH [48] and MIT Alewife [2] machines. A key component of these machines is the coherence controller on each node that provides cache coherent access to memory that is distributed among the nodes of the multiprocessor. In DASH and Alewife, the cache coherence protocol is hardwired in custom hardware finite state machines (FSMs) within the coherence controllers. Instead of hardwiring protocol handlers, the Sun Microsystems S3.mp [66] multiprocessor uses hardware sequencers for modularity in implementing protocol handlers.

Subsequent designs for scalable shared-memory multiprocessors, such as the Stanford FLASH [44] and the Wisconsin Typhoon machines [73], have touted the

use of programmable protocol processors instead of custom hardware FSMs to implement the coherence protocols. Although a custom hardware design generally yields better performance than a protocol processor for a particular coherence protocol, the programmable nature of a protocol processor allows one to tailor the cache coherence protocol to the application [17; 64], and may lead to shorter design times since protocol errors may be fixed in software. The study of the performance advantage of custom protocols is beyond the scope of this dissertation.

Performance simulations of the Stanford FLASH and Wisconsin Typhoon systems find that the performance penalty of protocol processors is small. Simulations of the Stanford FLASH, which uses a customized protocol processor optimized for handling coherence actions, show that the performance penalty of its protocol processor in comparison to custom hardware controllers is within 12% for most of the benchmarks considered [24]. Simulations of the Wisconsin Typhoon Simple-COMA system, which uses a protocol processor integrated with the other components of the coherence controller, also show competitive performance—within 30% of custom-hardware CC-NUMA controllers [73] and within 20% of custom-hardware Simple-COMA controllers [74].

Even so, the choice between custom hardware and protocol processors for implementing coherence protocols remains a key design issue for scalable shared-memory multiprocessors. In this chapter we examine in detail the performance tradeoffs between these two alternatives in designing a CC-NUMA multiprocessor

coherence controller. We consider symmetric multiprocessor (SMP) nodes as well as uniprocessor nodes as the building blocks for a multiprocessor. The availability of cost-effective SMPs, such as those based on the Intel Pentium Pro [35] makes SMP nodes an attractive choice for CC-NUMA designers [50]. However, the added load presented to the coherence controller by multiple SMP processors may affect the choice between custom hardware FSMs and protocol processors.

We base our experimental evaluation of alternative coherence controller architectures on realistic hardware parameters for state-of-the-art system components. What distinguishes our work from previous research is that we consider commodity protocol processors on SMP-based CC-NUMA and a wider range of architectural parameters. We simulate eight applications from the SPLASH-2 benchmark suite [93] to compare the application performance of the architectures. The results show that for a 64-processor system based on four-processor SMP nodes, protocol processors result in a performance penalty (increase in execution time relative to that of custom hardware controllers) of 4% – 93%.

The unexpectedly high penalty of protocol processors occurs for applications that have high-bandwidth communication requirements, such as Ocean, Radix, and FFT. The use of SMP nodes exacerbates the penalty. Previous research did not encounter such high penalties because the investigators were either comparing customized protocol processors in uniprocessor nodes, or they did not consider such high-bandwidth applications. We find that under high bandwidth require-

ments, the high occupancy of the protocol processor significantly degrades performance relative to custom hardware.

We also study the performance of coherence controllers with a pair protocol engines. Our results show that for applications with high communication requirements on a 4×16 CC-NUMA system, a two-engine hardware controller improves performance by up to 18% over a one-engine hardware controller, and a controller with two protocol processors improves performance by up to 30% over a controller with a single protocol processor

This study makes the following contributions:

- It provides an in-depth comparison of the performance tradeoffs between using custom hardware and protocol processors, and demonstrates situations where protocol processors suffer a significant penalty.

- It characterizes the communication requirements for eight applications from SPLASH-2 and shows the impact of those requirements on the performance penalty of protocol processors over custom hardware, and provides an understanding of application requirements and limitations of protocol processors.

- It evaluates the performance gains of using two protocol engines for custom hardware and protocol-processor-based coherence controllers.

- It introduces a methodology for predicting the impact of protocol engine implementation on the performance of important large applications through

the detailed simulation of simpler applications.

The rest of this chapter is organized as follows. Section 5.2 presents the multiprocessor system and details the controller design alternatives and parameters. Section 5.3 describes our experimental methodology and presents the experimental results. It demonstrates the performance tradeoffs and provides analysis of the causes of the performance differences between the architectures. Section 5.4 discusses related work. Finally, Section 5.5 presents the summary and conclusions drawn from this study and gives recommendations for custom hardware and protocol processor designs in future multiprocessors.

5.2 System Description

To put our results in the context of the architectures we studied, this section details these architectures and their key parameters. First we describe the organization and the key parameters of the common system components for the architectures. Then, we describe the details of the alternative coherence controller architectures. Finally, we present key protocol and coherence controller latencies and occupancies.

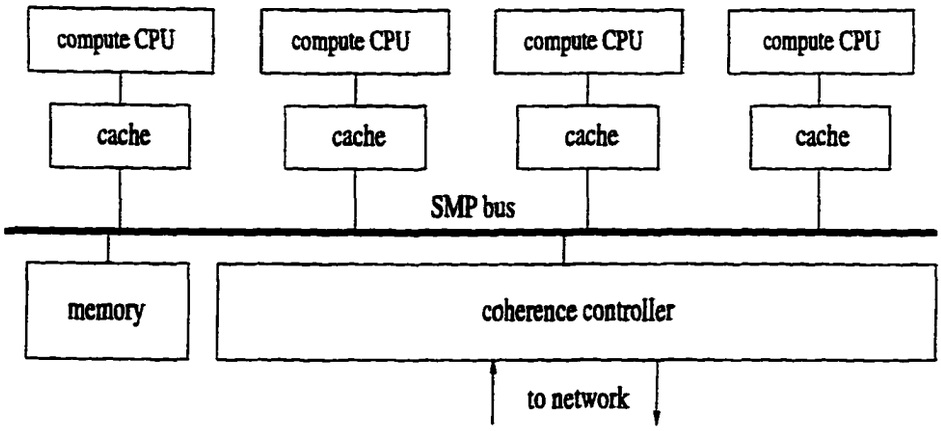

Figure 5.1: A node in a SMP-based CC-NUMA system.

5.2.1 General System Organization and Parameters

The base system configuration is a CC-NUMA multiprocessor composed of 16 SMP nodes connected by a 32 byte-wide fast state-of-the-art IBM switch. Each SMP node includes four 200 MHz PowerPC compute processors with 16 Kbyte L1 and 1 Mbyte L2 4-way-associative LRU caches, with 128 byte cache lines. The SMP bus is a 100 MHz 16 byte-wide fully-pipelined split-transaction separate-address-and-data bus. The memory is interleaved and the memory controller is a separate bus agent from the coherence controller. Figure 5.1 shows a block diagram of an SMP node. Table 5.1 shows the no-contention latencies of key system components. These latencies correspond to those of existing state-of-the-art components. Note that memory and cache-to-cache data transfers drive the critical quad-word first on the bus to minimize latency.

Event	Latency
L1 to processor	1
L1 to L2	8
L2 to L1	4
L2 miss to address strobe on bus	4
Bus address strobe to bus response	14
Bus address strobe to start of cache-to-cache data response	18
Bus address strobe to next address strobe	4
Bus address strobe to start of data transfer from memory	20
Network point-to-point	14

Table 5.1: Base system no-contention latencies in compute processor cycles (5 ns.).

5.2.2 Coherence Controller Architectures

We consider two main coherence controller designs: a custom hardware coherence controller similar to that in the DASH [48] and Alewife [2] systems, and a coherence controller based on commodity protocol processors similar to those in the Typhoon [73] system and its prototypes [74].

The two designs share some common components and features (see Figures 5.2 and 5.3). Both designs use duplicate directories to allow fast response to common requests on the pipelined SMP bus (one directory lookup per 2 bus cycles). The bus-side copy is abbreviated (2-bit state per cache line) and uses fast SRAM memory. The controller-side copy is full-bit-map and uses DRAM memory. Both designs use write-through directory caches for reducing directory read latency. Each directory cache holds up to 8K full-bit-map directory entries (e.g. approxi-

mately 16 Kbytes for a 16 node CC-NUMA system). The hardware-based design uses a custom on-chip cache, while the protocol-processor-based design uses the commodity processor's on-chip data caches.[1] We assume perfect instruction caches in the protocol processors, as the total size of all protocol handlers in our protocol is less than 16 Kbytes.

Both designs include a custom directory access controller for keeping the bus-side copy of the directory consistent with the controller-side copy, and a custom protocol dispatch controller for arbitration between the request queues from the local bus and the network. There are 3 input queues for protocol requests: bus-side requests, network-side requests, and network-side responses. The arbitration strategy between these queues is to let the network transaction nearest to completion be handled first. Thus, the arbitration policy is that network-side responses have the highest priority, then network-side requests, and finally bus-side requests. In order to avoid live-lock, the only exception to this policy is to allow bus-side requests which have been waiting for a long time (e.g. four subsequent network-side requests) to proceed before handling any more network-side requests.

Figure 5.2 shows a block diagram of a custom hardware coherence controller design (HWC). The controller runs at 100 MHz, the same frequency as the SMP bus. All the coherence controller components are on the same chip except the directories. Figure 5.3 shows a block diagram of a protocol-processor-based co-

[1]Although most processors use write-back caches, current processors (e.g. Pentium Pro [35]) allow users to designate regions of memory to be cached write-through.

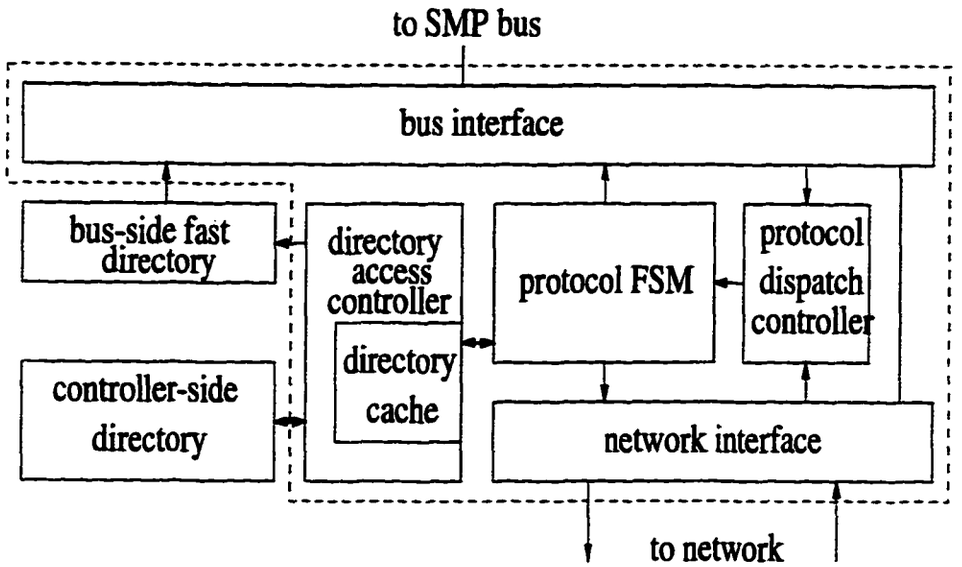

Figure 5.2: A custom-hardware-based coherence controller design (HWC).

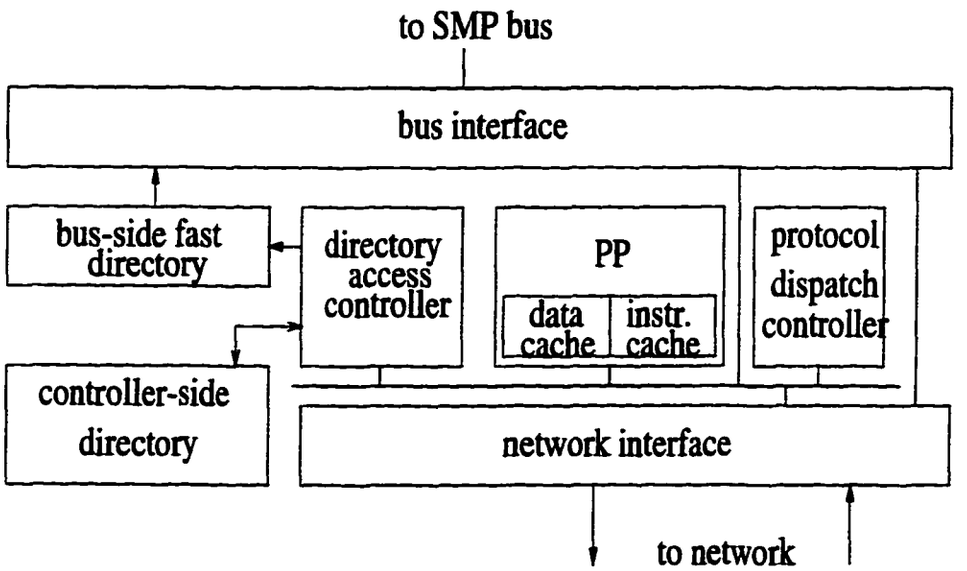

Figure 5.3: A commodity PP-based coherence controller design (PPC).

herence controller (PPC). The protocol processor (PP) is a PowerPC running at 200 MHz. The other controller components run at 100 MHz. The protocol processor communicates with the other components of the controller through loads

and stores on the local (coherence controller) bus to memory-mapped off-chip registers in the other components. The protocol processor access to the protocol dispatch controller register is read-only. Its access to the network interface registers is write-only (for sending network messages and starting direct data transfer from the bus interface), since reading the headers of incoming network messages is performed only by the protocol dispatch controller.

Both HWC and PPC have a direct data path between the bus interface and the network interface. The direct data path is used to forward write-backs of dirty remote data from the SMP bus directly to the network interface to be sent to the home node without waiting for protocol handler dispatch. Also, in the case of PPC, the PP only needs to perform a single write to a special register on either the bus interface or the network interface to invoke direct data transfer, without the need for the PP to read and write the data to perform the transfer.

In order to increase the bandwidth of the coherence controller we also consider the use of two protocol processors in the PPC implementation and two protocol FSMs in the HWC implementation. We use the term "protocol engine" to refer to both the protocol processor in the PPC design and the protocol FSM in the HWC design. For distributing the protocol requests between the two engines, we use a policy similar to that used in the S3.mp system [66], where protocol requests for memory addresses on the local node are handled by one protocol engine (LPE) and protocol requests for memory addresses on remote nodes are handled by the other

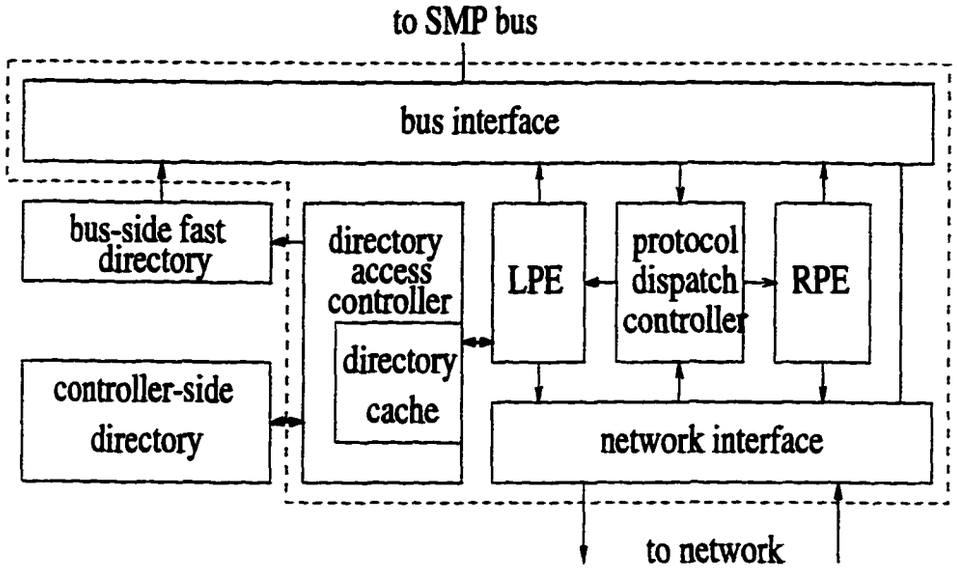

Figure 5.4: A custom hardware coherence controller design with local and remote protocol FSMs (2HWC).

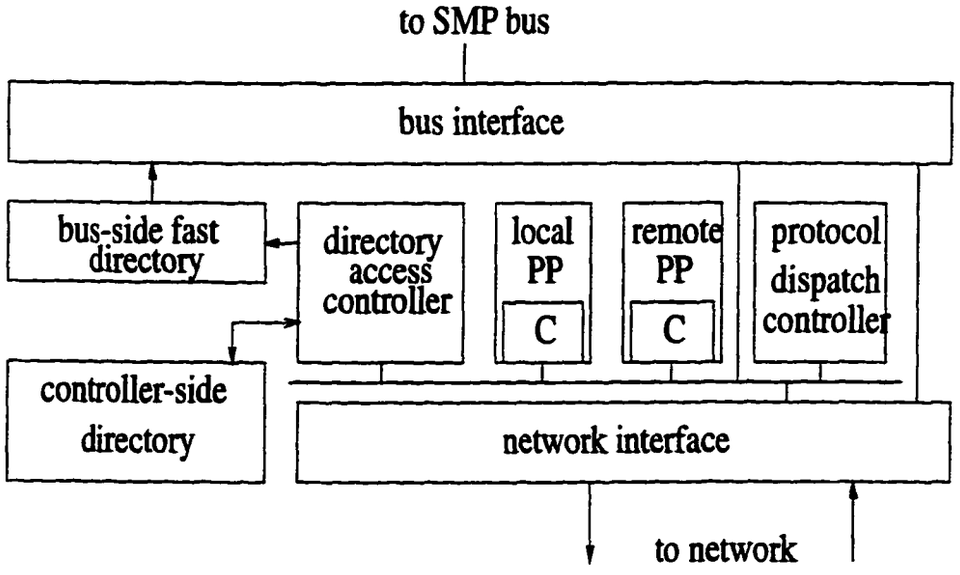

Figure 5.5: A commodity PP-based coherence controller design with local and remote protocol processors (2PPC).

Sub-operation	HWC	PPC
Issue request to bus	2	8
Detect response from bus	2	8
Issue network message	2	9
Read special bus interface associative registers	4	10
Write special bus interface registers	2	4
Directory read (cache hit)	2	2
Directory read (cache miss)	22	22
Directory write	2	2
Handler dispatch	2	12
Condition	2	2
Loop (per iteration)	2	5
Clear bit field	–	3
Extract bit field	–	2
Other bit operations	–	1

Table 5.2: Protocol engine sub-operation occupancies for HWC and PPC in compute processor cycles (5 ns.).

protocol engine (RPE). Only the LPE needs to access the directory. Figures 5.4 and 5.5 show the two-engine HWC design (2HWC), and the two PP controller design (2PPC), respectively.

5.2.3 Controller Latencies and Occupancies

We modeled HWC and PPC accurately with realistic parameters. Table 5.2 lists protocol engine sub-operations and their occupancies[2] for each of the HWC and PPC coherence controller designs, assuming a 100 MHz HWC and a 100 MHz PPC with a 200 MHz off-the-shelf protocol processor. The occupancies in

[2]Occupancy of sub-operations is the time a protocol engine is occupied by the sub-operation and cannot service other requests.

Step	HWC	PPC
detect L2 miss	8	8
issue bus read request	4	4
bus response	14	14
dispatch handler	2	12
extract home id	–	2
send message to home node	2	9
network latency	14	14
dispatch handler	2	12
read directory entry (cache hit)	2	2
conditions	2	6
issue bus read request	6	12
memory latency	20	20
detect bus response	2	8
extract requester's id	–	2
send message to the requester	2	9
network latency	14	14
dispatch handler	2	12
issue response to bus	6	12
L2 reissues read request	18	18
bus response	14	14
bus interface issues data	4	4
L1 fill	4	4
total	142	212

Table 5.3: Breakdown of the no-contention latency of a read miss to a remote line clean at home in compute processor cycles (5 ns.).

the table assume no contention on the SMP bus, memory, and network, and all directory reads hit in the protocol engine data cache. The other assumptions used in deriving these numbers are:

- Accesses to on-chip registers for HWC take one system cycle (2 CPU cycles).
- Bit operations on HWC are combined with other actions, such as conditions

and accesses to special registers.

- PP reads to off-chip registers on the local PPC bus take 4 system cycles (8 CPU cycles). Searching a set of associative registers takes an extra system cycle (2 CPU cycles).
- PP writes to off-chip registers on the local PPC bus take 2 system cycles (4 CPU cycles) before the PP can proceed.
- PP compute cycles are based on the PowerPC instruction cycle counts produced by the IBM XLC C compiler.
- HWC can decide multiple conditions in one cycle.

To gain insight into the effect of these occupancies and delays on the latency of a typical remote memory transaction, Table 5.3 presents the no-contention latency breakdown of a read miss from a remote node to a clean shared line at the home node. The relative increase in latency from HWC to PPC is only 49%, which is consistent with the 33% increase reported for Typhoon [74], taking into account that we consider a more decoupled coherence controller design and we use a faster network than that used in the Typhoon study. It is worth noting that in Table 5.3 there is no entry for updating the directory state at the home node. The reason is that updating the directory state can be performed after sending the response from the home node, thus minimizing the read miss latency. In our

protocol handlers, we postpone any protocol operations that are not essential for responding to requests until after issuing responses.

Finally, in order to gain insight into the relative occupancy times of the HWC and PPC coherence controller designs, Table 5.4 presents the no-contention protocol handler occupancies for the most frequently used handlers. Handler occupancy times include: handler dispatch time, directory reference time, access time to special registers, SMP bus and local memory access times, and bit field manipulation for PPC. Note that we use the same full-map directory, invalidation-based, write-back protocol, for both HWC and PPC. In our protocol, we allow remote owners to respond directly to remote requesters with data, but invalidation acknowledgments are collected only at the home node.

5.3 Experimental Results

In this section, we present simulation results of the relative performance of the different coherence controller architectures with several variations of communication-related architectural parameters. Then, we present analysis of the key statistics and communication measures collected from these simulations, and we conclude this section by presenting statistics and analysis of the utilization and workload distribution on two-protocol-engine coherence controllers. We start with the experimental methodology.

Handler	HWC	PPC
bus read remote	4	23
bus read exclusive remote	4	23
bus read local (dirty remote)	10	33
bus read excl. local (cached remote)	10 + 4/inv.	32 + 16/inv.
remote read to home (clean)	38	73
remote read to home (dirty remote)	10	29
remote read excl. to home (uncached remote)	38	73
remote read excl. to home (shared remote)	10 + 4/inv.	32 + 16/inv.
remote read excl. to home (dirty remote)	10	30
read from remote owner (request from home)	32	81
read from remote owner (remote requester)	34	90
read excl. from remote owner (request from home)	32	81
read excl. from remote owner (remote requester)	34	90
data response from owner to a read request from home	8	21
write back from owner to home in response to a read req. from remote node	8	24
data response from owner to a read excl. request from home	6	16
ack. from owner to home in response to a read excl. request from remote node	4	17
invalidation request from home to sharer	26	49
inv. acknowledgment (more expected)	8	23
inv. ack. (last ack, local request)	10	33
inv. ack. (last ack, remote request)	36	75
data in response to a remote read request	4	16
data in response to a remote read excl. request	6	20

Table 5.4: Protocol engine occupancies in compute processor cycles (5 ns.).

5.3.1 Experimental Methodology

We use execution-driven simulation (based on a version of the Augmint simulation toolkit [65] that runs on the PowerPC architecture) to evaluate the performance of the four coherence controller designs, HWC, PPC, 2HWC, and 2PPC.

Our simulator includes detailed contention models for SMP buses, memory controllers, interleaved memory banks, protocol engines, directory DRAM, and external point contention for the interconnection network. Protocol handlers are simulated at the granularity of the sub-operations in Table 5.2, in addition to accurate timing of the interaction between the coherence controller and the SMP bus, memory, directory, and network interface. All coherence controller implementations use the same cache coherence protocol.

We use eight benchmarks from the SPLASH-2 suite [93], (Table 5.5) to evaluate the performance of the four coherence controller implementations. All the benchmarks are written in C and compiled using IBM XLC C compiler with optimization level -O2. All experimental results reported in this study are for the parallel phase only of these applications. We use a round-robin page placement policy except for FFT where we use an optimized version with programmer hints for optimal page placement. We observed slightly inferior performance for most applications when we used a first-touch-after-initialization page placement policy, due to load imbalance, and memory and coherence controller contention as a result of uneven memory distribution. LU and Cholesky are run on 32-processor systems (8 nodes $\times$ 4 processors each) as they suffer from load imbalance on 64 processors with the data sets used [93]. We ran all the applications with data sizes and systems sizes for which they achieve acceptable speedups.

Application	Type	Problem size
LU	Blocked dense linear algebra	512×512 matrix, 16x16 blocks
Water-Spatial	Study of forces and potentials of water molecules in a 3-D grid	512 molecules
Barnes	Hierarchical N-body	8K particles
Cholesky	Blocked sparse linear algebra	tk15.O
Water-Nsquared	$O(n^2)$ study of forces and potentials in water molecules	512 molecules
Radix	Radix sort	1M integer keys, radix 1K
FFT	FFT computation	64K complex doubles
Ocean	Study of ocean movements	258×258 ocean grid

Table 5.5: Benchmark types and data sets.

5.3.2 Performance Results

In order to capture the main factors influencing PP performance penalty (the increase in execution time on PPC relative to the execution time on HWC), we ran experiments on the base system configuration with the four coherence controller architectures. We then varied some key system parameters to investigate their effect on the PP performance penalty.

Base Case

Figure 5.6 shows the execution times for the four coherence controller architectures on the base system configuration normalized by the execution time of

HWC. We notice that the PP penalty can be as high as 93% for Ocean and 52% for Radix, and as low as 4% for LU. The significant PP penalties for Ocean, Radix and FFT indicate that commodity PP-based coherence controllers can be the bottleneck when running communication-intensive applications. This result is in contrast to the results of previous research, which showed the cases where custom PP-based coherence controllers suffer small performance penalties relative to custom hardware.

Also, we observe that for applications with high bandwidth requirements, using two protocol engines improves performance significantly relative to the corresponding single engine implementation, up to 18% on HWC and 30% on PPC, for Ocean.

We varied other system and application parameters that are expected to have a big impact on the communication requirements of the applications. We start with the cache line size.

Smaller cache line size

With 32 byte cache lines, we expect the PP penalty to increase from that experienced with 128 byte cache lines, especially for applications with high spatial locality, due to the increase in the rate of requests to the coherence controller. Figure 5.7 shows the execution times normalized to the execution time of HWC on the base configuration. We notice a significant increase in execution time

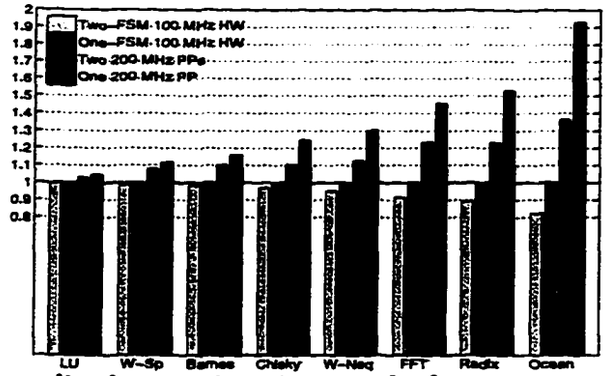

Figure 5.6: Normalized execution time on the base system configuration.

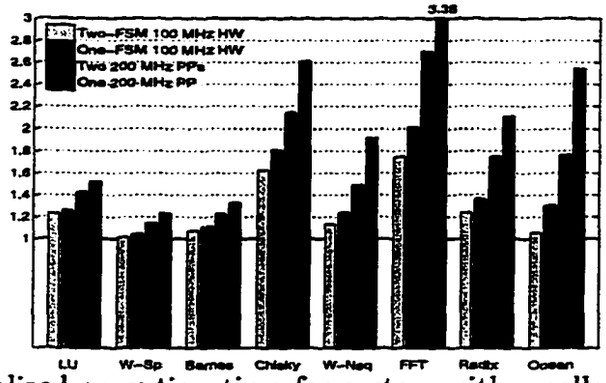

Figure 5.7: Normalized execution time for system with smaller (32 byte) cache lines.

(regardless of the coherence controller architecture) relative to the corresponding execution times on the base system, for FFT, Cholesky, Radix, and LU, which have high spatial locality [93], and a minor increase in execution time for the other benchmarks.

Also, we notice a significant increase in the PP penalty (compared to the PP penalty on the base system) for applications with high spatial locality, due to the increase in the number of requests to the coherence controllers, which increases the demand on PP occupancy. For example, the PP penalty for FFT increases from 45% to 68%.

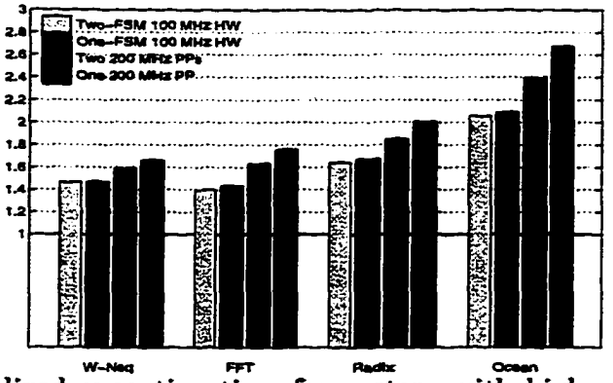

Figure 5.8: Normalized execution time for system with higher (1 μs.) network latency.

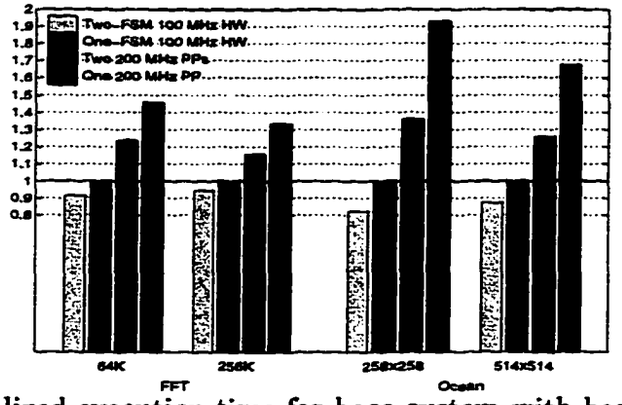

Figure 5.9: Normalized execution time for base system with base and large data sizes.

Slower network

To determine the impact of network speed on the PP performance penalty, we simulated the four applications with the largest PP penalties on a system with a slower network (1 μs. latency). Figure 5.8 shows the execution times normalized to the execution time of HWC on the base configuration. We notice a significant decrease in the PP penalty from that for the base system. The PP penalty for Ocean drops from 93% to 28%. Accordingly, systems designs with slow networks can afford to use commodity protocol processors instead of custom hardware,

without significant impact on performance, when cache line size is large.

Also, we notice a significant increase in execution time (regardless of the coherence controller architecture) relative to the corresponding execution times on the base system, for Ocean and Radix, due to their high communication rates.

Larger data size

To determine the effect of data size on the PP penalty, we simulated Ocean and FFT on the base system with larger data sizes, 256K complex doubles for FFT, and a 514×514 grid for Ocean. Figure 5.9 shows the execution times normalized by the execution time of HWC for each data size. We notice a decrease in the PP penalty in comparison to the penalty with the base data sizes, since the communication-to-computation ratios for Ocean and FFT decrease with the increase of the data size.[3] The PP penalty for FFT drops from 46% to 33%, and for Ocean from 93% to 67%.

However, since communication rates for applications like Ocean increase with the number of processors at the same rate that they decrease with larger data sizes, we can think of high PP performance penalties as limiting the scalability of such applications on systems with commodity PP-based coherence controllers.

[3]Applications like Radix maintain a constant communication rate with different data sizes [93].

Number of processors per SMP node

Varying the number of processors per SMP node (i.e. per coherence controller), proportionally varies the demand on the coherence controller occupancy, and thus is expected to impact the PP performance penalty. Figure 5.10 shows the execution times on 64-processor systems (32 for LU and Cholesky) with 1, 2, 4, and 8 processors per SMP node, normalized to the execution time of HWC on the base configuration (4 processors/node).

We notice that for applications with low communication rates, the increase in the number of processors per node has only a minor effect on the PP performance penalty. For applications with high communication rates, the increase in the number of processors increases the PP performance penalty (e.g. the PP penalty increases from 93% for Ocean on 4 processors per node to 106% on 8 processors per node). However, the PP penalty can be as high as 79% (for Ocean) even on systems with one processor per node.

For each of the coherence controller architectures, performance of applications with high communication rates degrades with more processors per node, due to the increase in occupancy per coherence controller, which are already critical resources on systems with fewer processors per node.

Also, we observe that for applications with high communication rates (except FFT), the use of two protocol engines in the coherence controllers achieves similar or better performance than controllers with one protocol engine with half the

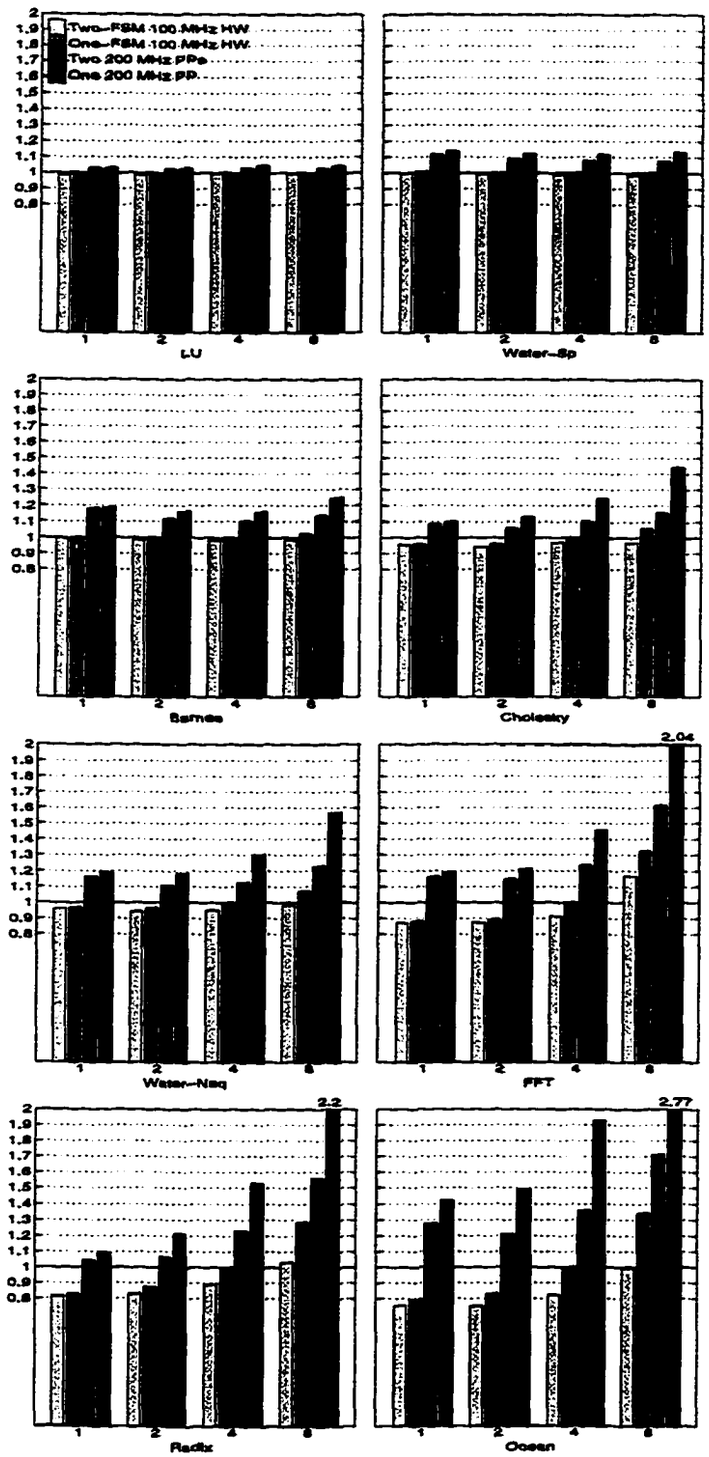

Figure 5.10: Normalized execution time with 1,2,4, and 8 processors per SMP node.

Application	PP Penalty	1000 x RCCPI	PPC/HWC occupancy	HWC utilization	PPC utilization	HWC queuing delay (ns.)	PPC queuing delay (ns.)	Average requests to HWC per μs.	Average requests to PPC per μs.
LU	4.37%	1.3	2.37	4.21%	9.58%	101	305	0.41	0.40
Water-Sp	11.69%	1.8	2.65	10.95%	25.99%	100	375	1.19	1.06
Barnes	15.81%	2.3	2.52	13.26%	28.88%	67	266	1.26	1.09
Cholesky	24.38%	4.1	2.23	26.38%	47.37%	113	365	2.34	1.86
Water-Nsq	30.15%	3.3	2.69	17.86%	36.87%	157	626	1.85	1.43
FFT-256K	33.44%	3.7	2.38	22.13%	39.54%	289	837	1.83	1.38
FFT-64K	45.59%	6.3	2.31	29.61%	46.96%	340	864	2.58	1.77
Radix	52.83%	9.8	2.36	36.82%	56.75%	229	640	3.66	2.33
Ocean-514	67.26%	14.0	2.29	47.54%	65.07%	226	648	3.87	2.31
Ocean-258	92.88%	23.2	2.47	52.89%	67.72%	232	720	4.69	2.41

Table 5.6: Communication statistics on the base system configuration.

number of processors per nodes. In other words, using two protocol engines in the coherence controllers, allows integrating twice as many processors per SMP node, thus saving the cost of half the SMP buses, memory controllers, coherence controllers, and I/O controllers.

5.3.3 Communication Statistics and Measures

In order to gain more insight into quantifying the application characteristics that affect PP performance penalty, we present some of the statistics generated by our simulations. Table 5.6 shows communication statistics collected from simulations of HWC and PPC on the base system configuration (except that Cholesky and LU are run on 32 processors). The statistics are:

- PP penalty: The increase in the execution time of PPC relative to the execution time of HWC.

- RCCPI (Requests to Coherence Controller Per Instruction): The total number of requests to the coherence controllers divided by the total number of instructions.

- Relative occupancy: The total of the occupancies of all coherence controllers for PPC divided by that for HWC.

- Average HWC (PPC) utilization: The average HWC (PPC) occupancy divided by execution time.

- Average HWC (PPC) queuing delay: The average time a request to the coherence controller waits in a queue while the controller is occupied by other requests.

- Arrival rate of requests to HWC (PPC) per μs. (200 CPU cycles): Derived from the reciprocal of the mean inter-arrival time of requests to each of the coherence controllers.

In Table 5.6 we notice that as RCCPI increases, the PP performance penalty increases proportionally except for Cholesky. In the case of Cholesky, the high load imbalance inflates the execution time with both HWC and PPC. Therefore, the PP penalty which is measured relative to the execution time with HWC is less than the PP penalty of other applications with similar RCCPI but with better load balance.

Also, as RCCPI increases, the arrival rate of requests to the coherence controller per cycle for PPC diverges from that of HWC, indicating that the PPC has saturated, and that the coherence controller is the bottleneck for the base system configuration. This is also supported by the observation of the high utilization rates of HWC with Ocean, and of PPC with Ocean, Radix, and FFT, indicating that the coherence controller has saturated these cases, and it is the main bottleneck.

However, we notice that the queuing delays do not increase proportionally with the increase in RCCPI, since the queuing effect of the coherence controller behaves like a negative feedback system where the increase in RCCPI (the input) increases the queuing delay in proportion to the difference between the queuing delay and a saturation value, thus limiting the increase in queuing delay. Note that the high queuing delay for FFT is attributed to its bursty communication pattern [93].

Also, we observe that the ratio between the occupancy of PPC and the occupancy of HWC is more or less constant for the different applications, approximately 2.5.

Figure 5.11 plots the arrival rate of requests to each of the coherence controller architectures against RCCPI for all the applications on the base system configuration (except Cholesky and LU as they were run on 32 processors) including Ocean and FFT with large data sizes. The dotted lines show the trend for each

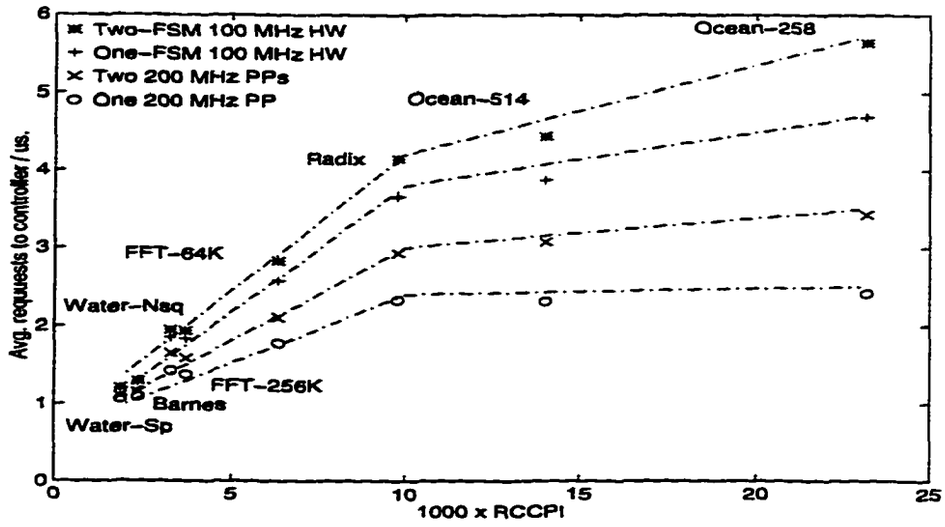

Figure 5.11: Coherence controller bandwidth limitations.

architecture. The figure shows clearly the saturation levels of the different coherence controller architectures. The divergence in the arrival rates demonstrates that the coherence architecture is the performance bottleneck of the base system.

Figure 5.12 shows the effect of RCCPI on the PP penalty for the same experiments as those in Figure 5.11. Figure 5.13 shows the effect of RCCPI on the relative performance of PPC vs. 2PPC with the same system configuration. Figures 5.14 and 5.15 show the effect of RCCPI on PP penalty for systems with 8 and 2 processors per node, respectively. We notice a clear proportional effect of RCCPI on the PP penalty. The gradual slope of the curve can be explained by the fact that the queuing model of the coherence controller resembles a negative feedback system. Without the negative feedback, the PP penalty would increase exponentially with the increase in RCCPI. The lower PP penalty for applications with low RCCPI such as Barnes and Water-Spatial is due to the fact that in those

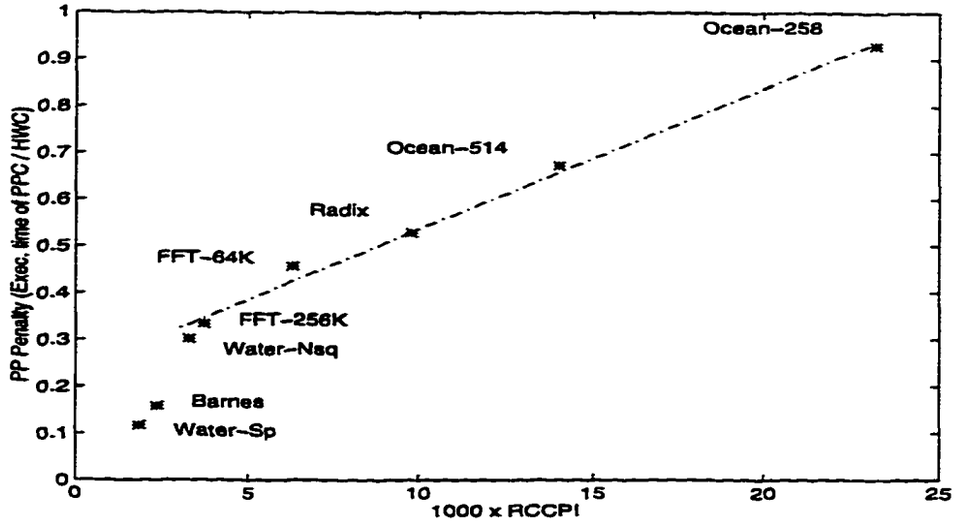

Figure 5.12: Effect of communication rate on PP penalty on the base system.

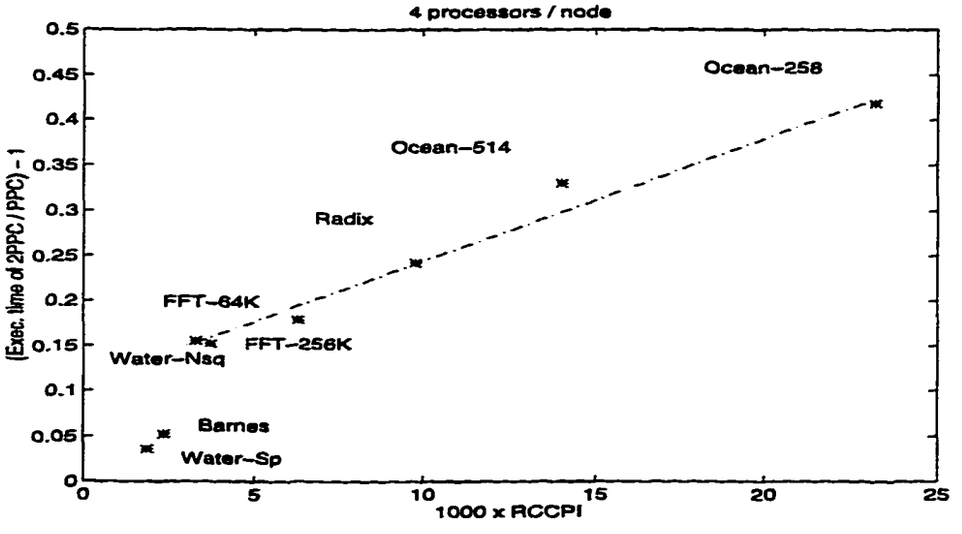

Figure 5.13: Effect of communication rate on the relative performance of PPC vs. 2PPC.

cases the coherence controller is under-utilized.

The previous analysis can help system designers predict the relative performance of alternative coherence controller designs. They can obtain the RCCPI measure for important large applications using simple simulators (e.g. PRAM)

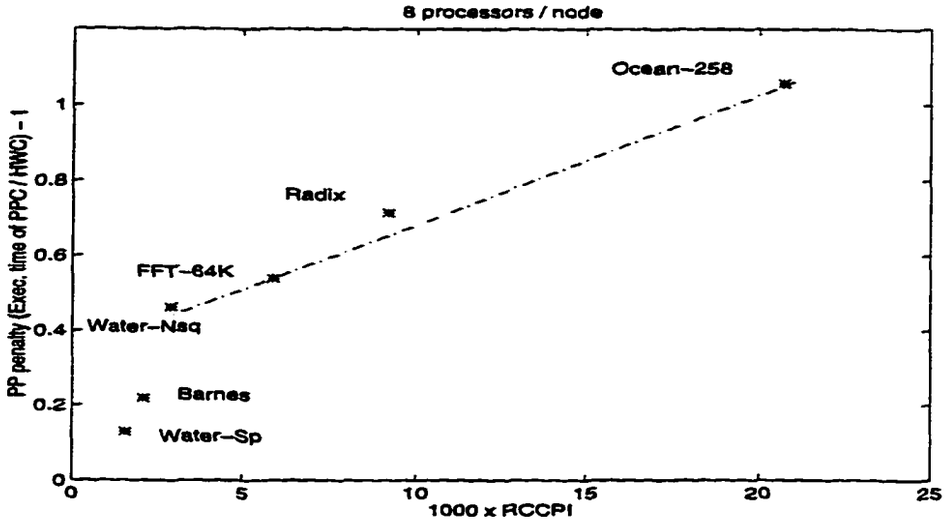

Figure 5.14: Effect of communication rate on PP penalty on a 8 processor/node system.

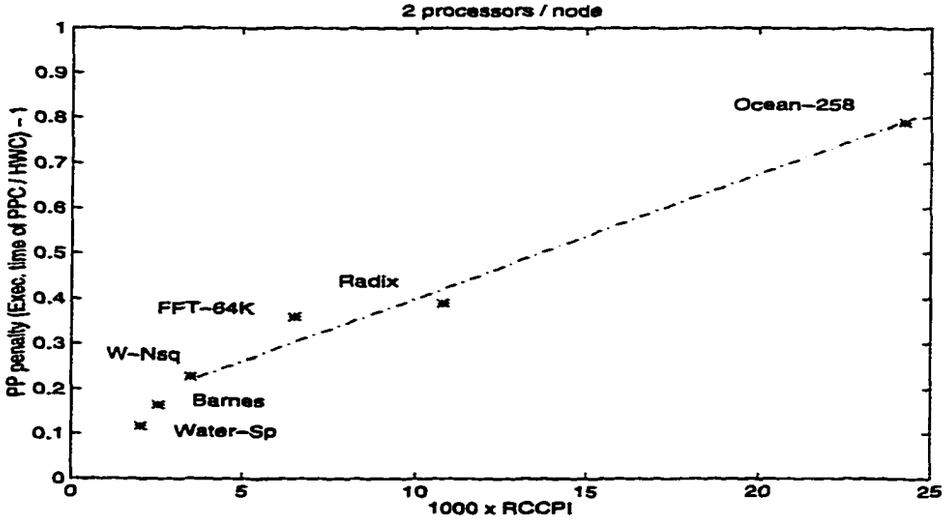

Figure 5.15: Effect of communication rate on PP penalty on a 2 processor/node system.

and relate that RCCPI to a graph similar to that in Figure 5.12 that can be obtained through the detailed simulation of simpler applications covering a range of communication rates similar to that of the large application. Although RCCPI is not necessarily independent of the implementation of the coherence controller, for

Application	Arch.	Utilization		Request distribution		Queuing delay (ns.)	
		LPE	RPE	LPE	RPE	LPE	RPE
LU	2HWC	3.20%	1.09%	35.67%	64.33%	182	2
	2PPC	5.66%	3.92%	35.74%	64.26%	501	14
Water-Sp	2HWC	6.82%	4.29%	38.09%	61.91%	60	40
	2PPC	14.66%	12.38%	38.08%	61.92%	263	78
Barnes	2HWC	8.43%	5.22%	39.38%	60.62%	67	11
	2PPC	16.64%	13.85%	39.41%	60.59%	237	53
Cholesky	2HWC	20.26%	7.48%	38.27%	61.73%	128	8
	2PPC	30.34%	19.99%	38.27%	61.73%	348	36
Water-Nsq	2HWC	11.30%	7.89%	39.26%	60.74%	82	49
	2PPC	22.87%	19.81%	39.22%	60.78%	384	167
FFT-256K	2HWC	17.93%	5.92%	46.33%	53.67%	378	10
	2PPC	30.64%	15.05%	46.33%	53.67%	934	38
FFT-64K	2HWC	25.63%	7.45%	41.40%	58.60%	478	8
	2PPC	36.35%	19.17%	41.40%	58.60%	1137	39
Radix	2HWC	21.63%	21.32%	39.95%	60.05%	138	91
	2PPC	30.70%	40.86%	39.94%	60.06%	243	366
Ocean-514	2HWC	38.10%	18.33%	41.03%	58.97%	210	35
	2PPC	50.42%	36.59%	41.02%	58.98%	480	138
Ocean-258	2HWC	40.02%	25.97%	40.45%	59.55%	173	48
	2PPC	52.60%	44.19%	40.39%	59.61%	476	185

Table 5.7: Communication statistics for controllers with two protocol engines on the base system configuration.

practical purposes the effect of the architecture on RCCPI can be ignored. In our experiments the difference in RCCPI between the four implementations (HWC, PPC, 2HWC, and 2PPC) is less than 1% for all applications.

5.3.4 Utilization of Two-Engine Controllers

For coherence controller architectures with two protocol engines, there is more than one way to split the workload between the two protocol engines. In this

study, we use a policy where protocol requests for memory addresses on the local node are handled by one protocol engine (LPE) and protocol requests for memory addresses on remote nodes are handled by the other protocol engine (RPE). In order to quantify the effectiveness of this policy, Table 5.7 shows the communication statistics collected from simulations of 2HWC and 2PPC on the base system configuration (except Cholesky and LU are run on 32 processors).

We observe that although most requests are handled by RPE (53-63%), the occupancy of LPE is up to 3 times that of RPE for 2HWC, and up to 2 times for 2PPC (derived from the utilization numbers). This is because the average occupancy of protocol handlers performed on LPE have higher average occupancy than those on RPE, since the former are more likely to access the directory and main memory. Also, we observe that the sum of the average utilization numbers for LPE and RPE is more than the average utilization for the corresponding one-engine coherence controller (Table 5.6). This is due to the fact that the sum of the occupancies of LPE and RPE is almost the same as that for the one-engine controller, but the execution time decreases with the use of two protocol engines.

Due to the imbalance between the utilization figures of LPE and RPE, the queuing delays for RPE are lower than those for the corresponding one-engine controllers, while those for LPE are higher for most applications despite the decrease in demand, due to the exclusion of the requests to RPE, which typically have low occupancy requirements.

The large imbalance in the distribution of occupancy between LPE and RPE (derived from the utilization statistics) for most applications indicates that there is potential for further improvement in performance by using a more even policy for distributing the workload on the two (or possibly more) protocol engines. However, it is worth noting that in the design used in this study, only one protocol engine, LPE, needs to access the directory. Furthermore, in the case of custom hardware, none of the handlers in the LPE FSM needs to be duplicated in the RPE FSM, and vice versa, thus minimizing the hardware overhead of two-engine HWC over one-engine HWC. Alternative distribution policies, such as splitting the workload dynamically or based on whether the request is from the local bus or another node, might lead to a more balanced distribution of protocol workloads on the protocol engines, but would also require allowing multiple protocol engines to access the directory, which increases the cost and complexity of coherence controllers.

5.4 Related Work

The proponents of protocol processors argue that the performance penalty of protocol processors is minimal, and that the additional flexibility is worth the performance penalty. The Stanford FLASH designers find that the performance penalty of using a protocol processor is less than 12% for the applications that they simulated, including Ocean and Radix [24]. Their measured penalties are

significantly lower than ours for the following reasons: 1) FLASH uses a protocol processor that is highly customized for executing protocol handlers, 2) they consider only uniprocessor nodes in their experiments, and 3) they assume a slower network latency of 220 ns., as opposed to 70 ns. in our base parameters.

In [73], Reinhardt *et al.* introduce the Wisconsin Typhoon architecture that relies on a SPARC processor core integrated with the other components of the coherence controller to execute coherence handlers that implement a Simple COMA protocol. Their simulations show that Simple COMA on Typhoon is less than 30% slower than a custom hardware CC-NUMA system. It is hard to compare our results to theirs because of the difficulty in determining what fraction of the performance difference is due to Simple COMA vs. CC-NUMA, and what fraction is due to custom hardware vs. protocol processors.

In [74], Reinhardt *et al.* compare the Wisconsin Typhoon and its first-generation prototypes with an idealized Simple COMA system. Here, their results show that the performance penalty of using integrated protocol processors is less than 20%. In contrast, we find larger performance penalties of up to 106%. There are two main reasons for this difference: 1) we are considering a more decoupled design than Typhoon, and 2) the application set used in the studies. Our results largely agree with theirs for Barnes, the only application in common between the two studies. However, we also consider applications with higher bandwidth requirements, such as Ocean, Radix, and FFT. Other differences between the two studies

are: a) they compare Simple COMA systems, while we compare CC-NUMA systems, b) they assume a slower network with a latency of 500 ns., which mitigates the penalty of protocol processors, and c) they consider only uniprocessor nodes.

Holt *et al.* [32] perform a study similar to ours. They also find that the occupancy of coherence controllers is critical to the performance of high-bandwidth applications. However, their work uses abstract parameters to model coherence controller performance, whereas our work considers practical, state-of-the-art controller designs. Also, our work provides strong insight into coherence controller bottlenecks, and we study the effect of having multiple processors per node and two protocol engines per coherence controller.

5.5 Summary

The major focus of this chapter has been to characterize the performance tradeoffs between using custom hardware versus protocol processors to implement cache coherence protocols. By comparing designs that differ only in features specific to either approach and keeping the rest of the architectural parameters identical, we were able to perform a systematic comparison of both approaches. We find that for applications with high bandwidth requirements, like Ocean, Radix, and FFT, the occupancy of off-the-shelf protocol processors significantly degrades performance by up to 106%. On the other hand, the programmable nature of protocol

processors allows one to tailor the cache coherence protocol to the application, and may lead to shorter design times since protocol errors may be fixed in software.

We also find that using a slow network or large data sizes results in tolerable protocol processor performance, and that for communication-intensive applications, performance degrades with the increase in the number of processors per node, as a result of the decrease in the number of coherence controllers in the system.

Our results also demonstrate the benefit of using two protocol engines in improving performance or maintaining the same performance of systems with larger number of coherence controllers. We are investigating other optimizations such as using more protocol engines for different regions of memory, and using custom hardware to implement accelerated data paths and handler paths for simple protocol handlers, which usually incur the highest penalties on protocol processors relative to custom hardware.

Our analysis of the application characteristics captures the communication requirements of the applications and their impact on performance penalty. Our characterization—RCCPI—can help system designers predict the performance of coherence controllers with other applications.

The results of our research imply that it is crucial to reduce protocol processor occupancy in order to support high-bandwidth applications. One approach is to custom design a protocol processor that is optimized for executing protocol

handlers, as in the Stanford FLASH multiprocessor. Another approach is to customize coherence protocols to the communication requirements of particular applications. We are currently investigating an alternative approach: to add incremental custom hardware to a protocol-processor-based design to accelerate common protocol handler actions.

6 Conclusions

6.1 Contributions

This dissertation contributes to improving the performance of shared memory multiprocessors, through characterizing and reducing the two main sources of overhead on shared memory multiprocessors: synchronization and cache coherence.

In more detail the contributions presented in this dissertation are:

- Presenting new algorithms:
  - An array-based priority queue heap that uses multiple mutual exclusion locks to allow consistent concurrent access [33]. The algorithm avoids deadlock among concurrent accesses without forcing insertions to proceed top-down. Bottom-up insertions reduce contention for the

topmost nodes of the heap, and avoid the need for a full-height traversal in many cases. The new algorithm also uses bit-reversal to increase concurrency among consecutive insertions, allowing them to follow mostly-disjoint paths. The new algorithm provides reasonable performance on small heaps, and significantly superior performance on large heaps under high levels of contention.

- A non-blocking shared link-based queue algorithm [60]. The algorithm is simple, practical, and fast. It seems to be the algorithm of choice for any queue-based application on a multiprocessor with universal atomic primitives (e.g. `compare_and_swap` or `load_linked/store_conditional`).
- A shared link-based queue algorithm with separate head and tail pointer locks [60]. The structure of the algorithm is similar to that of the non-blocking queue, but it allows only one enqueue and one dequeue to proceed at a given time. Because it is based on locks, however, it will work on machines with such simple atomic primitives as `test_and_set`. We recommend it for heavily-utilized queues on such machines.

- A demonstration that for simple data structures, special-purpose non-blocking atomic update algorithms will outperform all alternatives, not only on multiprogrammed systems, but on dedicated machines as well. Given the availability of a universal atomic hardware primitive, there seems to be no reason

to use any other version of a link-based stack, a link-based queue, or a small, fixed-sized object like a counter. For more complex data structures, however, or for machines without universal atomic primitives, preemption-safe locks are clearly important. Preemption-safe locks impose a modest performance penalty on dedicated systems, but provide dramatic savings on time-sliced systems. For the designers of future systems, we recommend (1) that hardware always include a universal atomic primitive, and (2) that kernel interfaces provide a mechanism for preemption-safe locking. [62]

- Implementation of the atomic primitives: `fetch_and_`Φ, `compare_and_swap`, and `load_linked/store_conditional`in the context of directory-based cache coherence protocols on DSM multiprocessors [59]. This study recommends implementing `compare_and_swap` with a write-invalidate cache coherence protocol in combination with a load exclusive primitive and supporting `fetch_and_clear_then_add` or `fetch_and_add` as an auxiliary primitive.

- Characterizing the performance tradeoffs between using custom hardware versus protocol processors to implement cache coherence protocols. [61] We find that for applications with high bandwidth requirements, the occupancy of off-the-shelf protocol processors significantly degrades performance by up to 106% for the applications we studied. On the other hand, the programmable nature of protocol processors allows one to tailor the cache coherence protocol to the application, and may lead to shorter design times

since protocol errors may be fixed in software. We also find that using a slow network or large data sizes results in tolerable protocol processor performance, and that for communication-intensive applications, performance degrades with the increase in the number of processors per node, as a result of the decrease in the number of coherence controllers in the system. Our results also demonstrate the benefit of using two protocol engines in improving performance or maintaining the same performance in systems with larger numbers of coherence controllers. Our analysis of application characteristics captures the communication requirements of the applications and their impact on the performance penalty of programmable controllers. Our characterization—RCCPI—can help system designers predict the performance of coherence controllers with other applications. Our results imply that it is crucial to reduce protocol processor occupancy in order to support high-bandwidth applications. One approach is to custom design a protocol processor that is optimized for executing protocol handlers, as in the Stanford FLASH multiprocessor. Another approach is to customize coherence protocols to the communication requirements of particular applications.

6.2 Future Directions

The `double_compare_and_swap` atomic primitive is very useful for implementing simple and fast non-blocking implementations of more complex data structures—

than those we presented in this dissertation—such as double-ended queues and doubly linked lists, or more efficient non-blocking data structures such circular queues [22; 54]. Extensive study of the design issues and alternatives of hardware implementations of `double_compare_and_swap` is required.

In coherence controller architectures with programmable protocol engines, the protocol processor can be some times underutilized. Executing application code, especially short critical sections, on protocol processors not only exploits underutilized protocol processors, but also might reduce the communication—and hence the demand on overutilized protocol processors—required for updating a shared data structure.

The performance measure, RCCPI, presented in Chapter 5 promises accurate prediction of large applications. Further study is needed for determining the applicability of this measure to different classes of applications than the scientific codes used in this dissertation. Also, it is important to develop mathematical models to capture the most significant factors in allowing RCCPI to predict performance.

Bibliography

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A Processor Architecture for Multiprocessing. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, May 1990.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[3] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, Vancouver, BC, Canada, August 1992.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *1990 ACM International Conference*

on Supercomputing, pages 1–6, Amsterdam, The Netherlands, June 1990. In *ACM SIGARCH Computer Architecture News 18*:3.

[5] 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.

[6] R. J. Anderson and H. Woll. Wait-Free Parallel Algorithms for the Union-Find Problem. In *Proceedings of the Twenty-Third ACM Symposium on Theory of Computing*, pages 370–380, May 1991.

[7] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[8] J. H. Anderson and M. Moir. Universal Constructions for Multi-Object Operations. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 184–194, Ottawa, Ontario, Canada, August 1995.

[9] G. Barnes. A Method for Implementing Lock-Free Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, Velen, Germany, June–July 1993.

[10] J. Biswas and J. C. Browne. Simultaneous Update of Priority Structures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 124–131, St. Charles, IL, August 1987.

[11] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, 23(5):35–43, May 1990.

[12] S. Bunch, R. Hochsprung, and T. Moore. The PowerPC Common Hardware Reference Platform: A System Architecture. In *Proceedings of the IEEE COMPCON '96*, Santa Clara, CA, February 1996.

[13] R. P. Case and A. Padegs. Architecture of the IBM System 370. *Communications of the ACM*, 21(1):73–96, January 1978.

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

[15] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, September 1988.

[16] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 1989.

[17] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings, Supercomputing'94*, Washington, DC, November 1994.

[18] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, MA, April 1989.

[19] A. Gottlieb and C. P. Kruskal. Coordinating Parallel Procesors: A Parallel Unification. *ACM SIGARCH Computer Architecture News*, 9(6):16–24, October 1981.

[20] 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.

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

[22] M. Greenwald and D. Cheriton. Practical Non-Blocking Synchronization Techinques for Operating System Software. In *Proceedings of the Second*

Symposium on Operating Systems Design and Implementation, Seattle, WA, October 1996.

[23] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(2):10–22, February 1992.

[24] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. P. Singh, R. Simoni, K. Gharachorloo, J. Baxter, D. Nakahira, M. Horowitz, A. Gupta, M. Rosemblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, San Jose, CA, October 1994.

[25] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[26] M. P. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 276–290, Toronto, Ontario, Canada, August 1988.

[27] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM Symposium on Principles*

and Practice of Parallel Programming, pages 197–206, Seattle, WA, March 1990.

[28] 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.

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

[30] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[31] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. Expanded version available as CRL 92/07, December Cambridge Research Laboratory, December 1992.

[32] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. *The Effects of Latency, Occupance, and Bandwidth in Distributed Shared Memory Multiprocessors.* Stanford University, January 1995.

[33] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An Efficient

Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, 60(3):151–157, November 1996.

[34] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[35] *Pentium Pro Family Developer's Manual.* Intel Corporation, 1996.

[36] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[37] D. W. Jones. Concurrent Operations on Priority Queues. *Communications of the ACM*, 32(1):132–137, January 1989.

[38] G. Kane. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

[39] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, October 1991.

[40] Kendall Square Research. KSR1 Principles of Operation. Waltham MA, 1992.

[41] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. In *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[42] O. Krieger, M. Stumm, and R. Unrau. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:201–204, St. Charles, IL, August 1993. CRC Press.

[43] H. Kung and P. Lehman. Concurrent Manipulation of Binary Search Trees. *ACM Transactions on Database Systems*, 5(3):339–353, 1980.

[44] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The FLASH Multiprocessor. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[45] A. LaMarca. A Performance Evaluation of Lock-free Synchronization Protocols. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 130–140, Los Angeles, CA, August 1994.

[46] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[47] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on

B-Trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[48] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[49] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, San Jose, CA, October 1994.

[50] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the Twenty-Third International Symposium on Computer Architecture*, pages 308–317, Philadelphia, PA, May 1996.

[51] *MIPS R4000 Microprocessor User's Manual.* MIPS Computer Systems, Inc., 1991.

[52] P. Magnussen, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 165–171, Cancun, Mexico, April 1994.

[53] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on*

Operating Systems Principles, pages 110–121, Pacific Grove, CA, October 1991.

[54] H. Massalin and C. Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, 1991.

[55] J. M. Mellor-Crummey. Concurrent Queues: Practical Fetch-and-Φ Algorithms. TR 229, Computer Science Department, University of Rochester, November 1987.

[56] 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.

[57] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Williamsburg, VA, April 1991.

[58] M. M. Michael and M. L. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. TR 599, Computer Science Department, University of Rochester, December 1995.

[59] M. M. Michael and M. L. Scott. Implementation of Atomic Primitives on Distributed Shared-Memory Multiprocessors. In *Proceedings of the First In-*

ternational Symposium on High Performance Computer Architecture, pages 222–231, Raleigh, NC, January 1995.

[60] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.

[61] M. M. Michael, A. K. Nanda, B.-H. Lim, and M. L. Scott. Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors. In *Proceedings of the Twenty-Fourth International Symposium on Computer Architecture*, pages 219–228, Denver, CO, June 1997.

[62] M. M. Michael and M. L. Scott. Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the Eleventh International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

[63] J. Mohan. Experience with Two Parallel Programs Solving the Travelling Salesman Problem. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 191–193, 1983.

[64] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory

Machines. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[65] A.-T. D. Nguyen, M. M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *Proceedings of the 1996 IEEE International Conference on Computer Design*, October 1996.

[66] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, Oconomowoc, WI, August 1995.

[67] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, FL, October 1982.

[68] S. Prakash, Y.-H. Lee, and T. Johnson. A Non-Blocking Algorithm for Shared Queues using Compare-and-Swap. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II:68–75, St. Charles, IL, August 1991.

[69] S. Prakash, Y. H. Lee, and T. Johnson. Non-Blocking Algorithms for Concurrent Data Structures. Technical Report 91-002, Department of Computer and Information Sciences, University of Florida, 1991.

[70] S. Prakash, Y. H. Lee, and T. Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.

[71] M. J. Quinn and N. Deo. Parallel Graph Algorithms. *ACM Computing Surveys*, 16(3):319–348, September 1984.

[72] V. N. Rao and V. Kumar. Concurrent Access of Priority Queues. *IEEE Transactions on Computers*, 37(12):1657–1665, December 1988.

[73] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared-Memory. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 325–336, Chicago, IL, April 1994.

[74] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the Twenty-Third International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[75] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the Eleventh International Symposium on Computer Architecture*, pages 340–347, 1984.

[76] B. Samadi. B-Trees in a System with Multiple Users. *Information Processing Letters*, 5(4):107–112, October 1976.

[77] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Ontario, Canada, August 1995.

[78] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, March 1992.

[79] R. Sites. Operating Systems and Computer Architecture. In H. Stone, editor, *Introduction to Computer Architecture*, page chapter 12. Science Research Associates, second edition, 1980.

[80] R. L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[81] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[82] J. M. Stone. A Simple and Correct Shared-Queue Algorithm Using Compare-and-Swap. In *Proceedings, Supercomputing '90*, New York, NY, November 1990.

[83] J. M. Stone. A Non-Blocking Compare-and-Swap Algorithm for a Shared Circular Queue. In S. Txafestas and others, editors, *Parallel and Distributed*

Computing in Engineering Systems, pages 147–152. Elsevier Science Publishers, 1992.

[84] H. S. Stone. *High Performance Computer Architecture*. Addison-Wesley, 1993.

[85] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 1(5):58–71, November 1993.

[86] R. K. Treiber. Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center, April 1986.

[87] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, Litchfield Park, AZ, December 1989.

[88] J. Turek, D. Shasha, and S. Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222, Vancouver, BC, Canada, August 1992.

[89] J. D. Valois. Implementing Lock-Free Queues. In *Proceedings of the Seventh*

International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, October 1994.

[90] J. D. Valois. Lock-Free Data Structures. Ph. D. dissertation, Rensselaer Polytechnic Institute, May 1995.

[91] J. D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, August 1995.

[92] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January – February 1994.

[93] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[94] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.