JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **51**, 1–26 (1998)
ARTICLE NO. PC981446

Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors[1]

Maged M. Michael[2] and Michael L. Scott

Department of Computer Science, University of Rochester, Rochester, New York 14627-0226
E-mail: michael@watson.ibm.com, scott@cs.rochester.edu

Received January 14, 1997; revised December 27, 1997; accepted January 10, 1998

Most multiprocessors are multiprogrammed to achieve acceptable response time and to increase their utilization. Unfortunately, inopportune preemption may significantly degrade the performance of synchronized parallel applications. To address this problem, researchers have developed two principal strategies for a concurrent, atomic update of shared data structures: (1) *preemption-safe locking* and (2) *nonblocking* (lock-free) *algorithms*. Preemption-safe locking requires kernel support. Nonblocking algorithms generally require a universal atomic primitive such as `compare-and-swap` or `load-linked/store-conditional` and are widely regarded as inefficient.

We evaluate the performance of preemption-safe lock-based and nonblocking implementations of important data structures—queues, stacks, heaps, and counters—including nonblocking and lock-based queue algorithms of our own, in microbenchmarks and real applications on a 12-processor SGI Challenge multiprocessor. Our results indicate that our nonblocking queue consistently outperforms the best known alternatives and that data-structure-specific nonblocking algorithms, which exist for queues, stacks, and counters, can work extremely well. Not only do they outperform preemption-safe lock-based algorithms on multiprogrammed machines, they also outperform ordinary locks on dedicated machines. At the same time, since general-purpose nonblocking techniques do not yet appear to be practical, preemption-safe locks remain the preferred alternative for complex data structures: they outperform conventional locks by significant margins on multiprogrammed systems. © 1998 Academic Press

[1] This work was supported in part by NSF Grants CDA-94-01142 and CCR-93-19445 and by ONR Research Grant 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 8930).

[2] Current address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

Key Words: nonblocking; lock-free; mutual exclusion; locks; multiprogramming; concurrent queues; concurrent stacks; concurrent heaps; concurrent counters; concurrent date structures; `compare-and-swap`; `load-linked`; `store-conditional`.

1. INTRODUCTION

Shared data structures are widely used in parallel applications and multiprocessor operating systems. To ensure the consistency of these data structures, processes perform synchronized concurrent update operations, mostly using critical sections protected by mutual exclusion locks. 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 [42] 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 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 [29] 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 multiprocessor. Another alternative is hardware partitioning, 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 *nonblocking 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 *nonblocking* (also known as *lock-free*) if it guarantees that at least one process of those trying to update the data structure concurrently 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. Nonblocking 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[3] atomic primitive such as `compare-and swap`[4] or the pair `load-linked` and `store-conditional`,[5] 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 nonblocking 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. For queues, we present fast new nonblocking and lock-based algorithms [27]. Our experimental results, employing both microbenchmarks and real applications, on a 12-processor Silicon Graphics Challenge multiprocessor, indicate that our nonblocking queue algorithm outperforms existing algorithms under almost all circumstances. In general, efficient data-structure-specific nonblocking algorithms outperform both ordinary and preemption-safe lock-based alternatives, not only on time-sliced systems, but on dedicated machines as well [28]. 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 nonblocking techniques in detail; previous work indicates that they are highly inefficient, though they provide a level of fault tolerance unavailable with locks. Our contributions include:

- A simple, fast, and practical nonblocking queue algorithm that outperforms all known alternatives and should be the algorithm of choice for multiprocessors that support universal atomic primitives.

- A two-lock queue algorithm that allows one enqueue and one dequeue to proceed concurrently. This algorithm should be used for heavily contended queues or multiprocessors with nonuniversal atomic primitives such as `test-and-set` or `fetch-and-add`.

- An evaluation of the performance of nonblocking algorithms in comparison to preemption-safe and ordinary (preemption-oblivious) locking for queues, stacks,

[3] Herlihy [9] presented a hierarchy of nonblocking objects that also applies to atomic primitives. A primitive is at level n of the hierarchy if it can provide a nonblocking solution to a consensus problem for up to n processors. Primitives at higher levels of the hierarchy can provide nonblocking 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.

[4] `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. `Compare-and-swap` is supported on the Intel Pentium Pro and Sparc V9 architectures.

[5] `Load-linked` and `store-conditional`, proposed by Jensen *et al.* [15], 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/store-conditional` is supported by the MIPS II, PowerPC, and Alpha architectures.

heaps, and counters. The paper demonstrates the superior performance of data-structure-specific nonblocking algorithms on time-slicing as well as dedicated multiprocessor systems.

The rest of this paper is organized as follows. We discuss preemption-safe locking in Section 2 and nonblocking algorithms in Section 3. In Section 4, we discuss nonblocking queue algorithms and present two concurrent queue algorithms of our own. We describe our experimental methodology and results in Section 5. Finally, we summarize our conclusions and recommendations in Section 6.

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 [7] 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 predefined 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 yields 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 [21] 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 [29] 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.* [16] presented a set of spin-then-block alternatives that adjust the spin time based on past experience. Black's work on Mach [6] 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.* [4] also support recovery: when a processor is taken from an application process, another active process belonging to the same application is informed via software interrupt. If the preempted process was holding a lock, the interrupted process 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 nondeterministic 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.* [17] presented preemption-safe (or "scheduler-conscious") versions of the ticket lock, the MCS lock [24], and Krieger *et al.*'s reader–writer lock [18]. 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 out-perform queue-based locks.

3. NONBLOCKING ALGORITHMS

Several nonblocking 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.

3.1. General Nonblocking Methodologies

Herlihy [10] presented a general methodology for transforming sequential implementations of data structures into concurrent nonblocking 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 nonblocking implementation of a skew-heap-based priority queue. Alemany and Felten [1] and LaMarca [19] 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 [5] presented a general methodology in which processes record and timestamp their modifications to the shared object and cooperate whenever conflicts arise. Shavit and Touitou [32] presented *software transactional memory*, which implements a k-word `compare-and-swap` using `load-linked/store-conditional`. Also, Anderson and Moir [2] presented nonblocking methodologies for large objects that rely on techniques for implementing multiple-word `compare-and-swap` using `load-linked/store-conditional` and vice verse. Turek *et al.* [39] and

Prakash *et al.* [30] presented methodologies for transforming multiple lock concurrent objects into lock-free concurrent objects. Unfortunately, theperformance of nonblocking algorithms resulting from general methodologies is acknowledged to be significantly inferior to that of the corresponding lock-based algorithms [10, 19, 32].

Two proposals for hardware support for general nonblocking data structures have been presented: *transactional memory* by Herlihy and Moss [11] and the *Oklahoma update* by Stone *et al.* [37]. 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.

3.2. Data-Structure-Specific Nonblocking Algorithms

Treiber [38] proposed a nonblocking implementation of concurrent link-based stacks. It represents the stack as a singly linked list with a *Top* pointer. It uses `compare-and-swap` to modify the value of *Top* atomically. Commented pseudocode of Treiber's nonblocking stack algorithm is presented in Fig. 1. No performance results were reported for nonblocking stacks. However, Treiber's stack is very simple and can be expected to be quite efficient. 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 [41] proposed a nonblocking implementation of linked lists. Simple nonblocking centralized counters can be implemented trivially using a `fetch-and-add` atomic primitive (if supported by hardware), or a read—modify—check—write cycle using `compare-and-swap` or `load-linked/store-conditional`.

```
structure pointer_t     {ptr: pointer to node_t, count: unsigned integer}
structure node_t        {value: data type, next: pointer_t}
structure 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
        node→next.ptr = NULL                            # Set next pointer of node to NULL
        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
```

FIG. 1. Structure and operation of Treiber's nonblocking concurrent stack algorithm [38].

```
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
```

FIG. 2. A nonblocking concurrent counter using `load-linked` and `store-conditional`.

Figure 2 shows a nonblocking counter implementation using `load-linked/store-conditional`.

Massalin and Pu [22] presented nonblocking algorithms for array-based stacks, array-based queues, and linked lists. Unfortunately, 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. No practical nonblocking implementations for array-based stacks or circular queues have been proposed. The general methodologies can be used, but the resulting algorithms would be very inefficient. For these data structures lock-based algorithms seem to be the only option.

In the following section, we continue the discussion of data-structure-specific nonblocking algorithms, concentrating on queues. Our presentation includes two new concurrent queue algorithms. One is non-blocking; the other uses a pair of mutual exclusion locks.

4. CONCURRENT QUEUE ALGORITHMS

4.1. Discussion of Previous Work

Many researchers have proposed lock-free algorithms for concurrent queues. Hwang and Briggs [14], Sites [33], and Stone [34] presented lock-free algorithms based on `compare-and-swap`. These algorithms are incompletely specified; they omit important details such as the handling of empty or single-item queues or concurrent enqueues and dequeues. Lamport [20] presented a wait-free algorithm that allows only a single enqueuer and a single dequeuer.[6] Gottlieb *et al.* [8] and Mellor-Crummey [23] presented algorithms that are lock-free but not nonblocking: they do not use locking mechanisms, but they allow a slow process to delay faster processes indefinitely. Treiber [38] presented an algorithm that is nonblocking but inefficient: a dequeue operation takes time proportional to the number of the elements in the queue.

As mentioned above, Massalin and Pu [22] presented a nonblocking array-based algorithm based on `double-compare-and-swap`, a primitive available only on later members of the Motorola 68000 family of processors. Herlihy and Wing [12] presented an array-based algorithm that requires infinite arrays. Valois [40] presented an array-based algorithm that requires either an unaligned

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

`compare-and-swap` (not supported on any architecture) or a Motorola-like `double-compare-and-swap`.

Stone [35] presented a queue that is lock-free but nonlinearizable[7] and not nonblocking. It is nonlinearizable 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 nonblocking 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 presented [36] a nonblocking 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 [31] presented a linearizable nonblocking algorithm that uses a singly linked list to represent the queue with *Head* and *Tail* pointers. It uses `compare-and-swap` to enqueue and dequeue nodes at the tail and the head of the list, respectively. A process performing an enqueue or a dequeue operation first takes a snapshot of the data structure and determines if there is another operation in progress. If so it tries to complete the ongoing operation and then takes another snapshot of the data structure. Otherwise it tries to complete its own operation. The process keeps trying until it completes its operation.

Valois [40] presented a list-based nonblocking queue 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 [33]). Unfortunately, the algorithm allows the tail pointer to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or reusing 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 [26] race conditions in the memory management mechanism and the associated nonblocking queue algorithm.

[7] 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 [13].

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

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

FIG. 3. Structure and operation of a nonblocking concurrent queue.

successful compare-and-swap. This solution does not guarantee that the *ABA* problem will not occur, but 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 [23] requires no special precautions to a void 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.

4.2. *New Algorithms*

We present two concurrent queue algorithms inspired by ideas in the work described above. Both of the algorithms are simple and practical. One is nonblocking; the other uses a pair of locks. Figure 3 presents commented pseudocode for the nonblocking queue data structure and operations. The algorithm implements the queue as a singly linked list with *Head* and *Tail* pointers. *Head* always points to a

```
structure node_t      {value: data type, next: pointer to node_t}
structure 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
```

FIG. 4. Structure and operation of a two-lock concurrent queue.

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 and then reuse dequeued nodes, the dequeue operation ensures that *Tail* does not point to the dequeued node or to any of its predecessors.

To obtain consistent values of various pointers we rely on sequences of reads that recheck earlier values to be sure they have not 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. A simple and efficient nonblocking stack algorithm due to Treiber [38] can be used to implement a nonblocking free list.

Figure 4 presents commented pseudocode 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 nonblocking 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 deadlock problems that might arise from processes trying to acquire the locks in different order.

Experimental results comparing these algorithms with others are presented in Section 5. A discussion of algorithm correctness is presented in Appendix A.

5. EXPERIMENTAL RESULTS

We use a Silicon Graphics Challenge multiprocessor with twelve 100 MHz MIPS R4000 processors to compare the performance of the most promising nonblocking, ordinary lock-based, and preemption-safe lock-based implementations of counters and of link-based queues, stacks, and skew heaps. We use microbenchmarks 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.[8]

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 pseudoscheduler. Whenever a process is due for preemption, the pseudoscheduler interrupts it, forcing it into a signal handler. The handler spins on a flag which the pseudoscheduler 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.

[8] C code for all the microbenchmarks and the real applications are available from ftp://ftp.cs.rochester.edu/pub/packages/sched_conscious_synch/multiprogramming.

All ordinary and preemption-safe locks used in the experiments are test-and-test-and-set locks with bounded exponential backoff. All nonblocking algorithms also bounded exponential backoff. The effectiveness of backoff in reducing contention on locks and synchronization data is demonstrated in the literature [3, 24]. 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 Fig. 5.

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.

5.1. *Queues*

Figure 6 shows performance results for eight queue implementations on a dedicated system (no multiprogramming), and on multiprogrammed systems with two and three 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 nonblocking algorithm (**MS nonblocking**) and those due to Prakash *et al.* [31] (**PLJ nonblocking**) and Valois [40] (**Valois nonblocking**); and Mellor-Crummey's blocking algorithm [23] (**MC blocking**). We include the algorithm of Prakash *et al.* because it appears to be the best of the known nonblocking 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 inefficient nonblocking algorithm can outperform blocking algorithms.

```
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
```

FIG. 5. Implementations of `test-and-set` and `compare-and-swap` using `load-linked` and `store-conditional`.

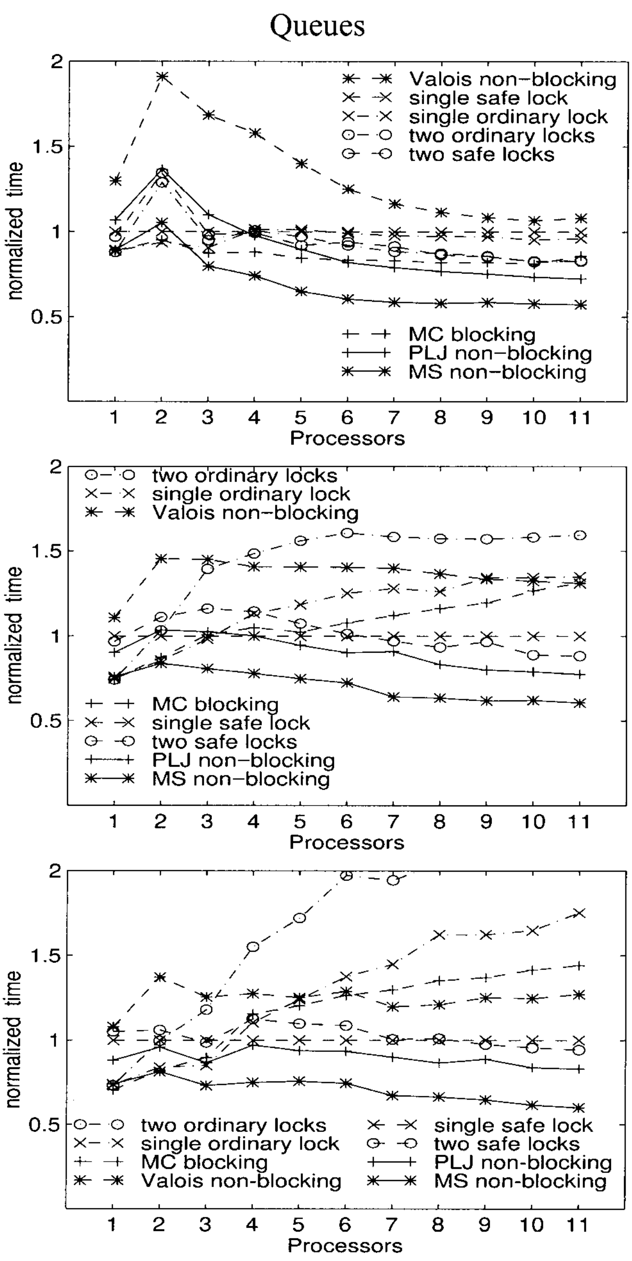

FIG. 6. Normalized execution time for 1,000,000 enqueue/dequeue pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

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 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 one, two, and three 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 nonblocking 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 processors. This effect is more obvious in the multiple lock and nonblocking 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 four processors are active in our microbenchmark. With multiprogramming levels of 2 and 3, the crossover points for the one- and two-lock algorithms with preemption-safe locks occur at six and eight processors, respectively. The nonblocking 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.

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 nonblocking 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.

5.2. Stacks

Figure 7 shows performance results for four stack implementations on a dedicated system and on multiprogrammed systems with two and three processes

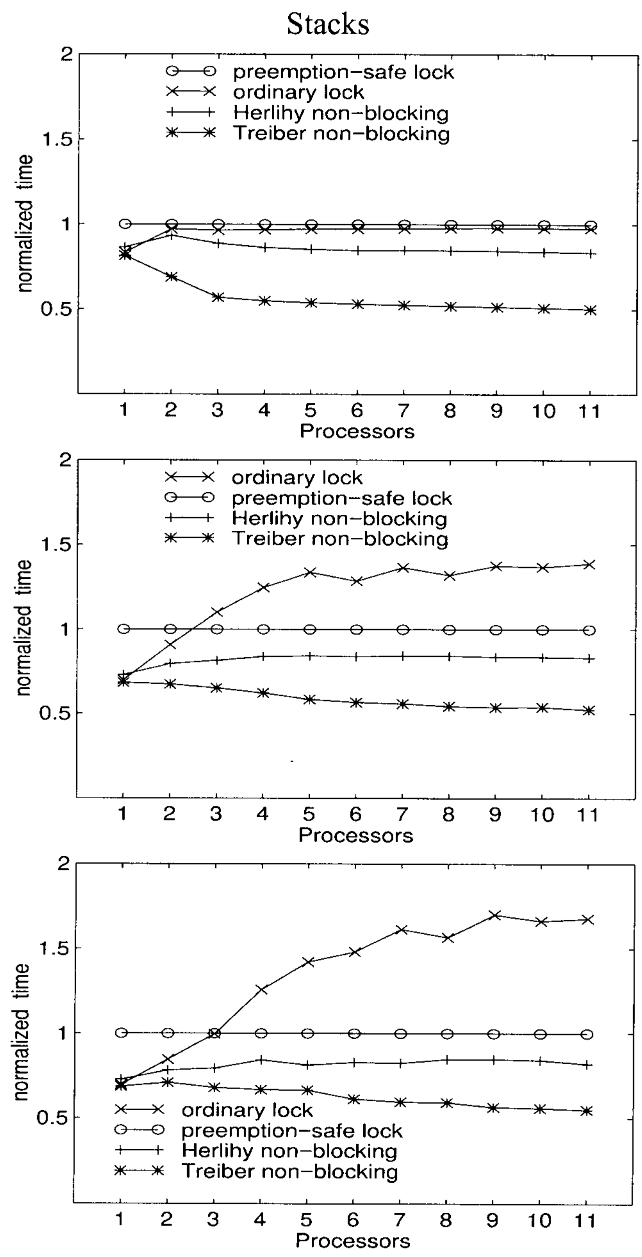

FIG. 7. Normalized execution time for 1,000,000 push/pop pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

per processor. The four stack implementations are the usual single-lock algorithm using ordinary and preemption-safe locks, Treiber's nonblocking stack algorithm [38], and an optimized nonblocking algorithm based on Herlihy's general methodology [10].

Like Treiber's nonblocking 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, within one, two, and three processes 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 nonblocking 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 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.

5.3. *Heaps*

Figure 8 shows performance results for three skew heap implementations on a dedicated system and on multiprogrammed systems with two and three processes per processor The three implementations are the usual single-lock algorithm using ordinary and preemption-safe locks and an optimized nonblocking algorithm due to Herlihy [10].

The optimized nonblocking 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 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 one, two, and three 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 nonempty 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 nonblocking algorithms remains relatively unchanged. The degradation of the ordinary locks is

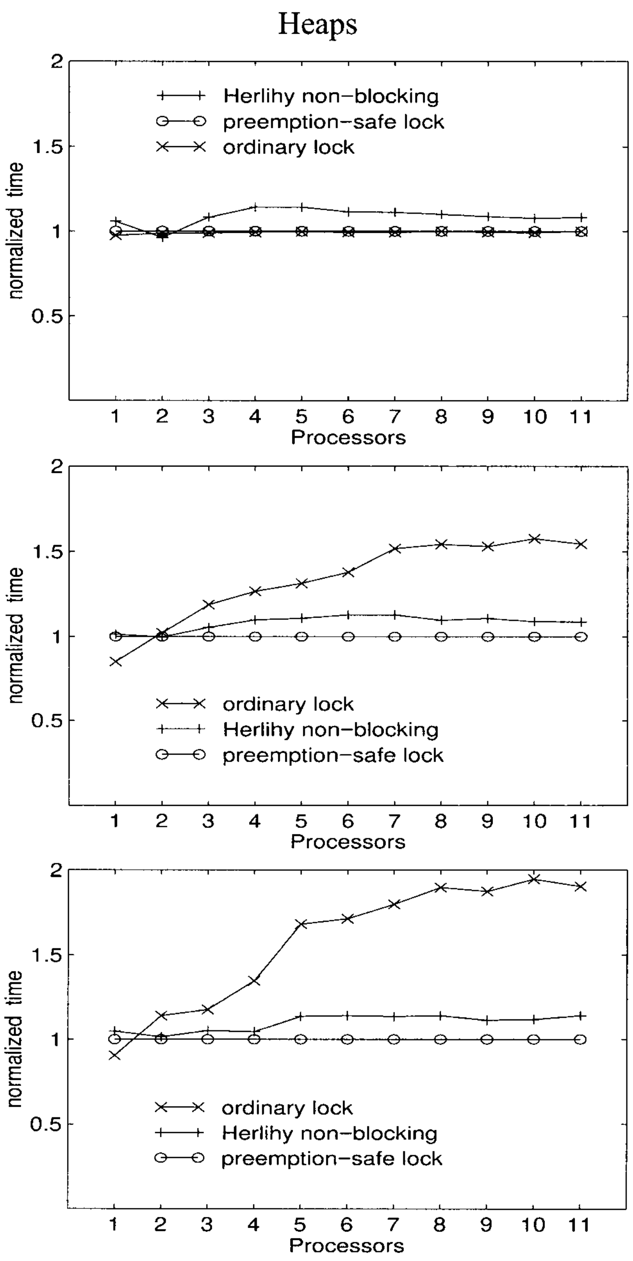

FIG. 8. Normalized execution time for 1,000,000 insert/delete_min pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

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 nonblocking 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 nonblocking methodologies (without data-structure-specific elimination of copying) are even more complex and could be expected to perform much worse.

5.4. *Counters*

Figure 9 shows performance results for three implementations of counters on a dedicated system and on multiprogrammed systems with two and three processes per processor. The three implementations are the usual single-lock algorithm using ordinary and preemption-safe locks and the nonblocking 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 one, two, and three 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 nonblocking algorithm outperforms the preemption-safe lock-based counter by more than 55% on 11 processors with various levels of multiprogramming. The performance of a `fetch-and-add` atomic primitive would be even better [25].

5.5. *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 for distributing 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 nonblocking queue and Treiber's stack, respectively. In each execution, the processes cooperate in sorting an array of 500,000 pseudorandom numbers using quicksort for intervals of more than 20 elements and insertion sort for smaller intervals.

Figure 10 shows performance results for the three queue-based versions; Fig. 11 shows 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 one, two, and three 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 microbenchmarks. Performance with ordinary locks degrades under multiprogramming, though not as severely as before, since more work is being done between atomic operations. Simple nonblocking algorithms yield superior performance even on dedicated systems, making them the algorithm of choice under any level of contention or multiprogramming.

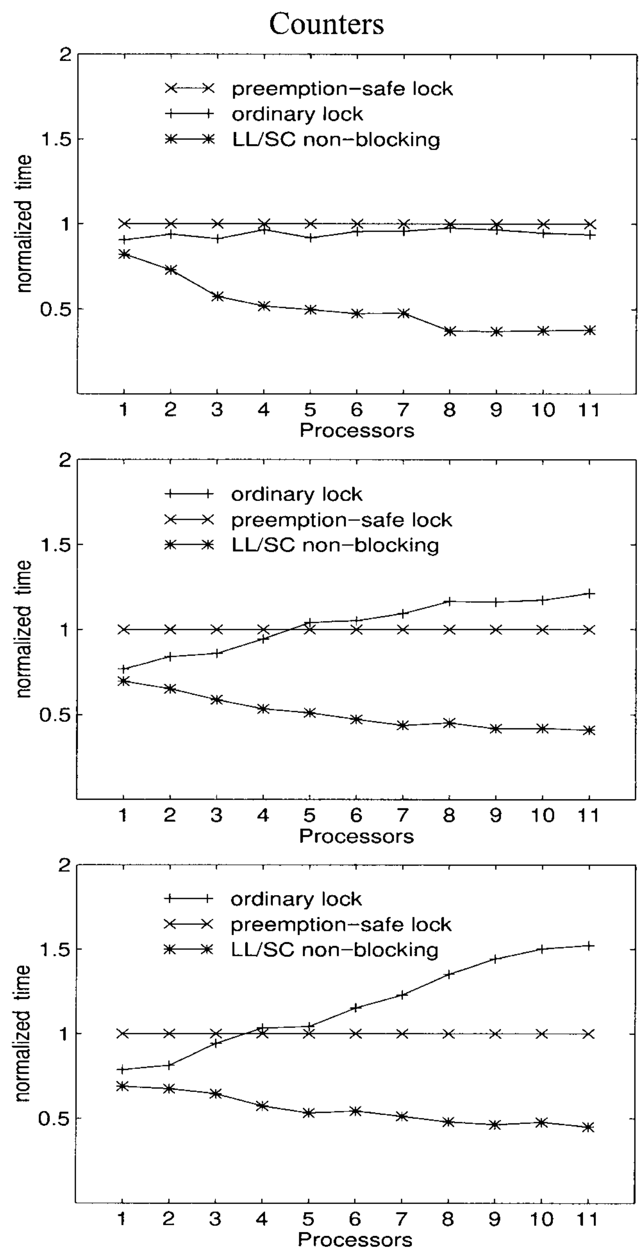

FIG. 9. Normalized execution time for 1,000,000 atomic increments on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

5.6. 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. 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

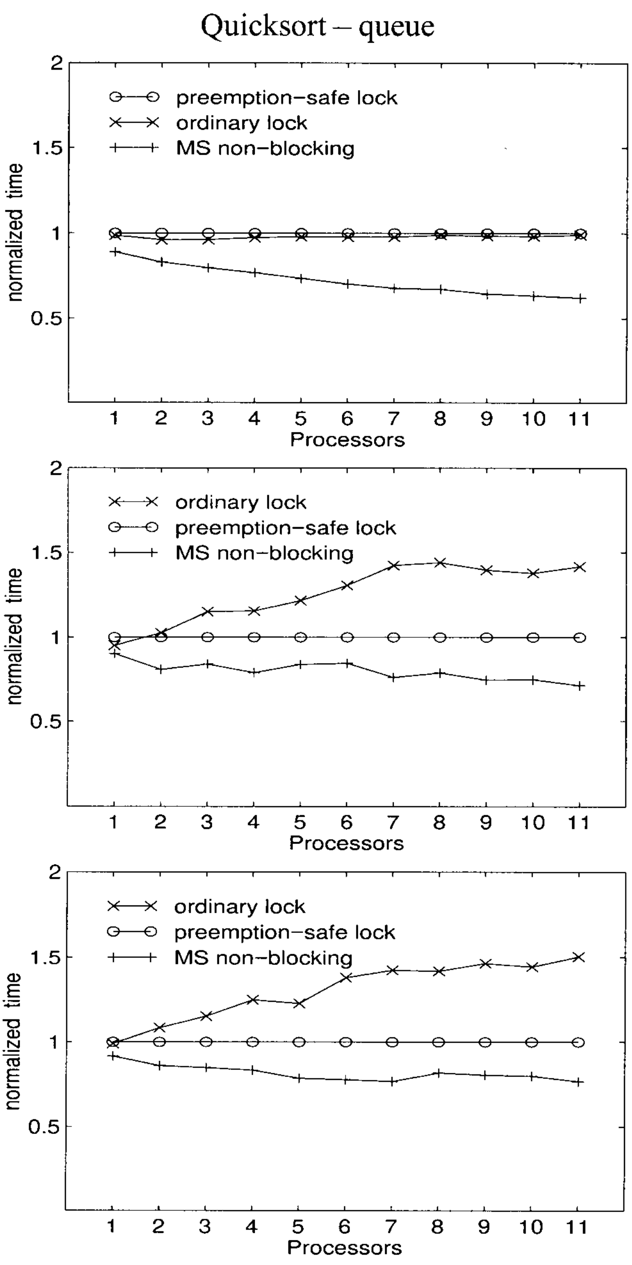

FIG. 10. Normalized execution time for quicksort of 500,000 items using a shared queue on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

nonblocking 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

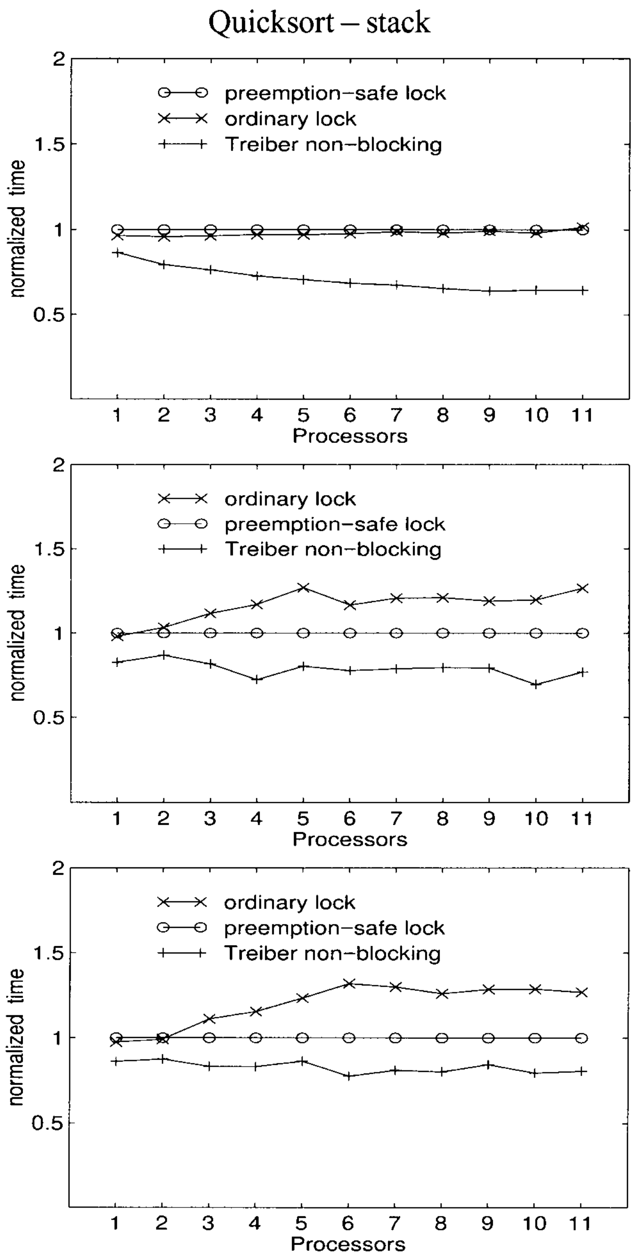

FIG. 11. Normalized execution time for quicksort of 500,000 items using a shared stack on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

that uses the version of each data structure that ran the fastest for the microbenchmarks: nonblocking stacks and counters and a preemption-safe priority queue.

Figure 12 shows 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

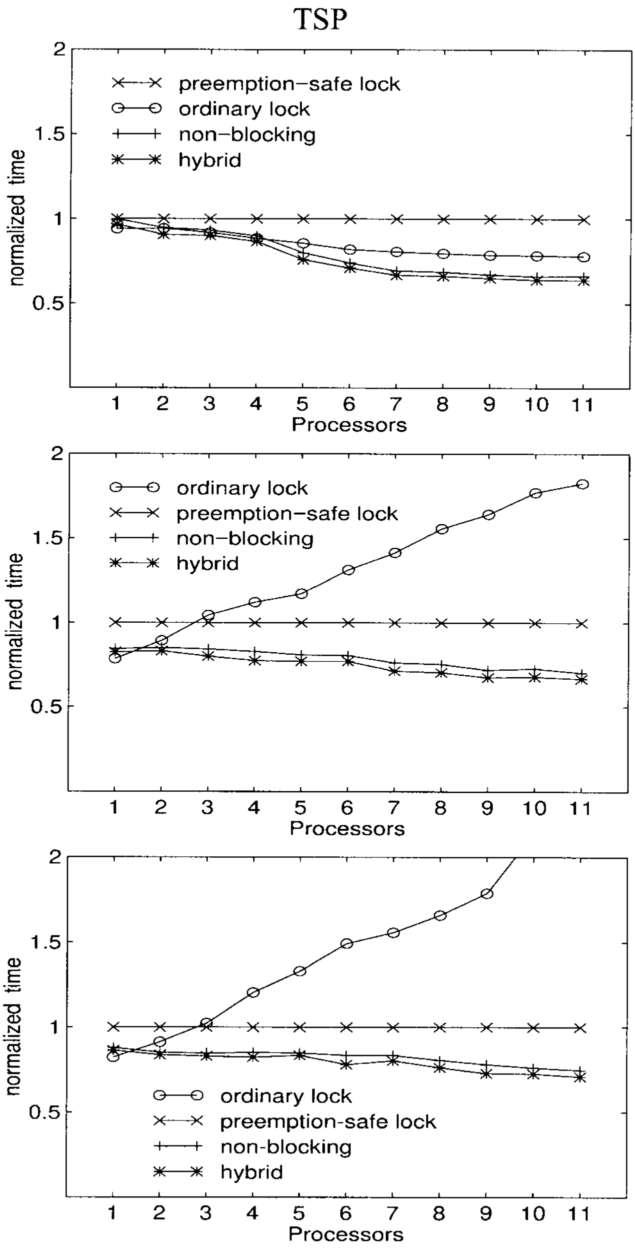

FIG. 12. Normalized execution time for a 17-city traveling salesman problem using a shared priority queue, stack and counters on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

and 11 processors, with one, two, and three 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 microbenchmarks, 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.

6. CONCLUSIONS

For atomic updates of a shared data structure, the programmer may ensure consistency using (1) a single lock, (2) multiple locks, (3) a general-purpose nonblocking technique, or (4) a special-purpose (data-structure-specific) nonblocking 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 multilock and nonblocking 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 nonblocking 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 such as 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. For small-scale machines, the Symunix interface [7] appears to work well. For larger machines, a more elaborate interface may be appropriate [17].

We have presented a concurrent queue algorithm that is simple, nonblocking, practical, and fast. It appears to be the algorithm of choice for any queue-based application on a multiprocessor with a universal atomic primitive. Also, we have presented a two-lock queue algorithm. Because it is based on locks, it will work on machines with such nonuniversal 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 perform better.

REFERENCES

1. 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, August 1992, pp. 125–134.

2. J. H. Anderson and M. Moir, Universal constructions for multi-object operations, *in* "Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing," Ottawa, Ontario, August 1995, pp. 184–194.

3. T. E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, *IEEE Trans. Parallel Distrib. Syst.* **1**, 1 (January 1990), 6–16.

4. 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 Trans. Comput. Syst.* **10**, 1 (February 1992), 53–79.

5. G. Barnes, A method for implementing lock-free data structures, *in* “Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures,” Velen, Germany, June–July 1993, pp. 261–270.

6. D. L. Black, Scheduling support for concurrency and parallelism in the mach operating system, *Computer* **23**, 5 (May 1990), 35–43.

7. 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.

8. A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors, *ACM Trans. Progrmg. Lang. Syst.* **5**, 2 (April 1983), 164–189.

9. M. P. Herlihy, Wait-free synchronization, *ACM Trans. Progrmg. Lang. Syst.* **13**, 1 (January 1991), 124–149.

10. M. P. Herlihy, A methodology for implementing highly concurrent data objects, *ACM Trans. Progrmg. Lang. Syst.* **15**, 5 (November 1993), 745–770.

11. M. P. Herlihy and J. E. Moss, Transactional memory: Architectural support for lock-free data structures, *in* “Proceedings of the Twentieth International Symposium on Computer Architecture,” San Diego, May 1993, pp. 289–300.

12. M. P. Herlihy and J. M. Wing, Axioms for concurrent object, *in* “Proceedings of the 14th ACM Symposium on Principles of Programming Languages,” January 1987, pp. 13–26.

13. M. P. Herlihy and J. M. Wing, Linearizability: A correctness condition for concurrent objects, *ACM Trans. Progrmg. Lang. Syst.* **12**, 3 (July 1990), 463–492.

14. K. Hwang and F. A. Briggs, “Computer Architecture and Parallel Processing,” McGraw–Hill, New York, 1984.

15. 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.

16. A. R. Karlin, K. Li, M. S. Manasse, and S. S. Owicki, Empirical studies of competitive spinning for a shared-memory multiprocessor, *in* “Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles,” Pacific Grove, CA, October 1991, pp. 41–55.

17. L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, Scheduler-conscious synchronization, *ACM Trans. Comput. Syst.* **15**, 1 (February 1997).

18. O. Krieger, M. Stumm, and R. C. Unrau, A fair fast scalable reader–writer lock, *in* “Proceedings of the 1993 International Conference on Parallel Processing,” St. Charles, IL, August 1993, pp. II: 201–204.

19. A. LaMarca, A performance evaluation of lock-free synchronization protocols, *in* “Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing,” Los Angeles, August 1994, pp. 130–140.

20. L. Lamport, Specifying concurrent program modulus, *ACM Trans. Progrmg. Lang. Syst.* **5**, 2 (April 1983), 190–222.

21. 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,” Pacific Grove, CA, October 1991, pp. 110–121.

22. H. Massalin and C. Pu, A lock-free multiprocessor OS kernel, Technical report CUCS-005-91, Computer Science Department, Columbia University, 1991.

23. J. M. Mellor-Crummey, Concurrent queues: Practical fetch-and-Φ Algorithms, TR 229, Computer Science Department, University of Rochester, November 1987.

24. J. M. Mellor-Crummey and M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Trans. Comput. Syst.* **9**, 1 (February 1991), 21–65.

25. M. M. Michael and M. L. Scott, Implementation of atomic primitives on distributed shared-memory multiprocessors, *in* “Proceedings of the First International Symposium on High Performance Computer Architecture, Raleigh, NC, January 1995,” pp. 222–231.

26. M. M. Michael and M. L. Scott, Correction of a memory management method for lock-free data structures, Technical Report 599, Computer Science Department, University of Rochester, December 1995.
27. 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," Philadelphia, May 1996, pp. 267–275.
28. 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.
29. J. K. Ousterhout, Scheduling techniques for concurrent systems, *in* "Proceedings of the Third International Conference on Distributed Computing Systems," October 1982, pp. 22–30.
30. S. Prakash, Y. H. Lee, and T. Johnson, Non-blocking algorithms for concurrent data structures, Technical Report 91-002, University of Florida, July 1991.
31. S. Prakash, Y. H. Lee, and T. Johnson, A nonblocking algorithm for shared queues using compare-and-swap, *IEEE Trans. Comput.* **43**, 5 (May 1994), 548–559.
32. N. Shavit and D. Touitou, Transactional memory, *in* "Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing," Ottawa, Ontario, August 1995, pp. 204–213.
33. R. Sites, Operating systems and computer architecture, *in* "Introduction to Computer Architecture" (H. Stone, Ed.), 2nd ed. Chap. 12, 1980. Science Research Associates, Chicago.
34. H. S. Stone, "High Performance Computer Architecture," Addison–Wesley, Reading, MA, 1993.
35. J. M. Stone, A simple and correct shard-queue algorithm using compare-and-swap, *in* "Proceedings of Supercomputing '90," Minneapolis, November 1990, pp. 495–504.
36. J. M. Stone, A non-blocking compare-and-swap algorithm for a shared circular queue, *in* "Parallel and Distributed Computing in Engineering Systems" (S. Tzafestas *et al*., Eds.), Elsevier Science, Amsterdam/New York, pp. 147–152, 1992.
37. J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, Multiple reservations and the Oklahoma update, *IEEE Parallel Distrib. Tech.* **1**, 5 (November 1993), 58–71.
38. R. K. Treiber, Systems programming: Coping with parallelism, "RJ 5118, Almaden Research Center," April 1986.
39. J. Turek, D. Shasha, and S. Prakash, Locking without blocking: Making lock based concurrent data structure algorithms nonblocking, *in* "Proceedings of the 11th ACM Symposium on Principles of Database Systems," Vancouver, BC, August 1992, pp. 212–222.
40. 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, pp. 64–69.
41. J. D. Valois, Lock-free linked lists using compare-and-swap, *in* "Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing," Ottawa, Ontario, August 1995, pp. 214–222.
42. J. Zahorjan, E. D. Lazowska, and D. L. Eager, The effect of scheduling discipline on spin overhead in shared memory parallel systems, *IEEE Trans. Parallel Distrib. Syst.* **2**, 2 (April 1991), 180–198.

MAGED M. MICHAEL is a research staff member in the Multiprocessor Communication Architecture Department at the IBM Thomas J. Watson Research Center. He received a Ph.D. in computer science from the University of Rochester in 1997. His research interests include shared memory multiprocessor synchronization, multiprocessor architecture, execution-driven simulation, and cache coherence on large-scale multiprocessors. He devised concurrent algorithms for mutual exclusion locks, barriers and shared data structures. He is a co-designer of the Augmint multiprocessor execution-driven simulation environment.

MICHAEL L. SCOTT is Chair of the Computer Science Department at the University of Rochester, where he co-leads the Cashmere shared memory project. He received his Ph.D. in computer sciences in 1985 from the University of Wisconsin—Madison. His research interests include operating systems, programming languages, and program development tools for parallel and distributed computing. His textbook on programming language design and implementation is scheduled to be published by Morgan Kaufmann in 1998. Other recent publications have addressed scalable synchronization algorithms and software cache coherence.