Evaluating and Designing Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors

Xiaodong Zhang, Yong Yan, and Robert Castañeda
High-Performance Computing and Software Laboratory, University of Texas at San Antonio

Performance evaluations of software-based mutual exclusion algorithms must take into account the effects of architectures and systems. The authors demonstrate a framework for such evaluation, and use the framework as a basis for designing more efficient algorithms.

Shared-memory multiprocessor systems must provide facilities called *critical sections* for programs to share physical and logical resources. Only one processor at a time can process a CS. This requires a method to ensure mutually exclusive access to the logically atomic operations of a shared CS.

Two approaches to mutual exclusion are hardware algorithms, which use primitives, and software algorithms, which require only software read and write instructions (see the "Hardware primitives versus software algorithms" sidebar).

We previously examined architecture and system effects on spin-locks, a type of hardware algorithm.[1] Extensive experiments on two different shared-memory multiprocessor systems—Bolt, Beranek, and Newman's TC2000 and Kendall Square Research's KSR-1—showed that the execution behavior of these algorithms, and therefore their performance, differs significantly. These machines use different types of interconnection networks and cache/memory systems. This construction produces different types of nonuniform memory access (NUMA) execution patterns for the mutual exclusion algorithms.

We've concluded that architecture and system effects should be seriously considered when developing and implementing mutual exclusion algorithms. Our experience has motivated us to investigate comprehensive performance effects on portable software mutual exclusion protocols.

However, standard analyses of mutual exclusion algorithms do not fully account for these effects. For example, these analyses express the memory-access complexity of an algorithm as a function of N, the number of

1063-6552/96/$4.00 © 1996 IEEE

Hardware primitives versus software algorithms

All shared-memory multiprocessor systems provide hardware support for allowing mutually exclusive access to critical variables. This hardware primitive usually consists of instructions that atomically read and then write a single memory location. Many efficient mutual exclusion algorithms have been constructed by using certain atomic hardware primitives. However, these algorithms require strong hardware support and are architecture-dependent.

Software-based mutual exclusion algorithms are an attractive alternative. Some complex architecture- or model-dependent phenomena can be understood only on the basis of empirical observations of program executions. Software mutual-exclusion algorithms could be ported to different platforms to evaluate these phenomena.

Efficient software mutual exclusion algorithms have other potential uses. First, these algorithms could effectively support synchronization primitives for implementing a distributed shared-memory layer on top of a message-passing system, such as a network of workstations where global spin locks are not available. Second, the unique flexibility of software-based algorithms could be used to take advantage of variations of shared-memory architectures. Multiple accesses to a lock can cause hot-spot contention in hardware-based schemes. Hierarchical structures in software-based algorithms could distribute these accesses.

Software-based mutual exclusion algorithms have major limitations. Their complex structure can cause significant implementation overhead. Also, each algorithm needs a rigorous proof of its correctness. However, we believe the feasibility of using these algorithms needs further investigation because no one has obtained sufficient experimental performance results to reach a consensus.

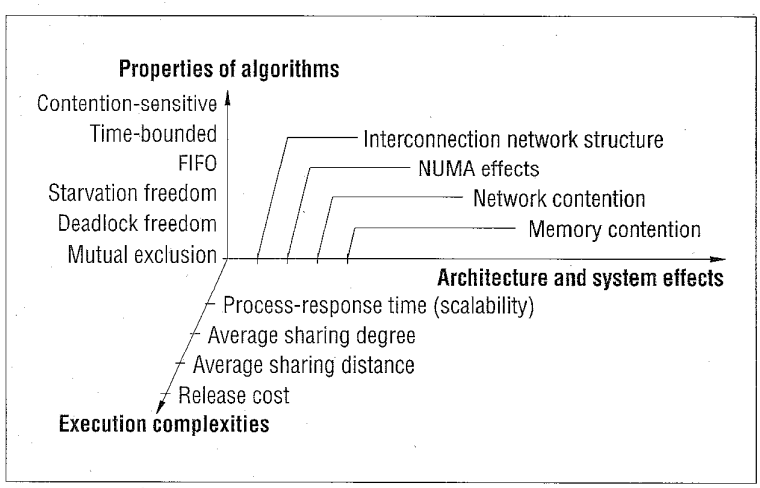

Figure 1. Performance-evaluation framework for mutual exclusion algorithms based on the properties of algorithms, execution complexities, and architecture and system effects.

We propose a comprehensive performance-evaluation framework that examines the overhead patterns inherent in the mutual exclusion algorithms and in the architectures on which the algorithms run. We used this framework to evaluate several representative mutual exclusion algorithms on the BBN TC2000 and KSR-1.

Our research with this framework has helped us determine characteristics of efficient software mutual exclusion algorithms. Based on these characteristics, we've developed three mutual exclusion algorithms, two of which combine good features of two of the representative algorithms. Tests show that these new algorithms are fast and can be highly scalable.

remote accesses to shared memory. This notation has a limited ability to represent the algorithm's complexity, for two reasons. First, in a modern NUMA shared-memory system, remote memory accesses can be constructed at more than one level, which can cause different delays for different accesses. Second, data-access distribution can strongly affect network contention. For example, the same number of hot-spot remote accesses and the same number of distributed remote accesses will create significantly different network contention overheads, which results in much lower performance for the hot-spot case. Algorithm analyses describing a general number of remote memory accesses will not fully and precisely express the overhead patterns in mutual exclusion algorithms or the effects of interconnection network architectures.

A performance-evaluation framework

Our framework is based on three performance factors:

1. properties of algorithms,
2. execution complexities of algorithms, and
3. architecture and system effects on algorithms.

Figure 1 gives a 3D view of the framework.

Properties of algorithms

A mutual exclusion algorithm has three basic properties:

- *Mutual exclusion* guarantees that only one process executes the CS at any time.
- *Deadlock freedom* guarantees that processes will not be blocked forever.

Early mutual exclusion algorithms

Considerable research has concentrated on software mutual exclusion algorithms. T. Deckker was the first to devise a software solution.[1] His solution was simply a combination of a strict alternative method and a lock to aid a process to enter the critical section (CS). The limitation of using a strict alternative solution is that a process that is in its noncritical section can block another process from entering the CS. Each process must wait its turn regardless of whether it needs to reenter the CS immediately. The limitation of using just a lock is that two processes might enter a race condition upon reading and then updating the lock. Thus, the two processes might enter the CS simultaneously.

G.L. Peterson proposed a simpler approach to achieving mutual exclusion.[2] In Peterson's algorithm, each process sets a software lock and shows its desire to enter the CS. This method does not have a strict alternative restriction and avoids the race condition. The number of accesses per CS invocation to the shared lock of each process is $\Omega(n)$, where n is the number of processes.

Research on software mutual exclusion algorithms for shared-memory systems has been active since 1987, when Leslie Lamport presented two algorithms in which a process makes a constant number of accesses to the shared lock only if no access contention occurs.[3] Lamport's first algorithm requires an upper bound on the time needed to perform an individual operation such as a memory reference, an upper bound on the time needed to execute the CS, or both. Because this algorithm must be fine-tuned for every machine on which it runs, it is not practical for solving the mutual exclusion problem. Lamport's second algorithm does not require time bounds. This approach is practical to use because it does not require any fine tuning for an upper bound to wait for an instruction or execution of the CS.

Lamport's solutions have spurred many new software solutions with other types of requirements, such as first-in, first-out service or contention sensitivity. However, no general algorithm appeals to all. A machine's architecture, and the memory organization on which the algorithm is implemented, can significantly affect the algorithm's performance.

References

1. E.W. Dijkstra, "Co-operating Sequential Processes," in *Programming Languages*, F. Genuys, ed., Academic Press, London, 1965, pp. 43–112.
2. G.L. Peterson, "Myths About the Mutual Exclusion Problem," *Information Processing Letters*, Vol. 12, No. 3, June 1981, pp. 115–116.
3. L. Lamport, "A Fast Mutual Exclusion Algorithm," *ACM Trans. Computer Systems*, Vol. 5, No. 1, 1987, pp. 1–11.

- *Starvation-freedom* guarantees that each requesting process eventually executes the CS.

Some additional properties impose stronger conditions on an algorithm, but are not required for the algorithm's correctness. (Stronger requirements might actually significantly increase the management overhead and slow down the mutual exclusion execution.) Three major properties are

- *FIFO*, which guarantees a fair order for each requesting process to access the CS. To maintain this property, higher overhead is inevitable. Because a shared-memory system does not have a global clock, additional software structures must form and maintain a FIFO process queue.
- *The time-bound condition*, which requires placement of an assumed time bound on the instruction speed or on the execution time of the CS. Based on this property, mutual exclusion algorithms fall into two categories, following Lamport's two algorithms (see the "Early mutual exclusion algorithms" sidebar). The first requires time bounds; the second does not. Algorithms that do not require time bounds are more useful in practice.
- *Contention sensitivity*, which relates to process dependence. The performance of a good contention-sensitive mutual exclusion algorithm should be affected only by the number of processes contending for the CS, not by the total number of executing processes.

Execution complexities of algorithms

Instead of theoretical memory-access complexity, we propose a set of execution complexities, which cover the effects of architecture, system, and software implementation on an algorithm. Execution complexities consist of four execution measurements and analyses: *process-response time*, *sharing degree*, *sharing distance*, and *release cost*.

Process-response time measures the average delay for a requesting process to access the CS; this delay indicates the algorithm's overall execution performance. The measured response time might not be consistent with the result of the algorithm's theoretical complexity using the total number of remote accesses. The following three measures provide further insights into the execution performance.

Sharing degree (S_{deg}) for a shared variable defines the maximum number of process threads concurrently accessing the variable. The function $S_{deg}(v)$ returns the sharing degree of variable v. This measure lets us examine shared-variable access distributions for an algorithm to detect hot spots (areas of strong contention, which we'll discuss in more detail later). The average sharing degree for an algorithm is

$$\bar{S}_{deg} = \frac{\sum_{i=1}^{k} S_{deg}(v_i)}{k},$$

where v_i for $(i = 1, ..., k)$ represents the shared variables in the algorithm.

Sharing distance (S_{dis}) for a shared variable defines the maximum remote-access time to the variable from a processor. The function $S_{dis}(v)$ returns the sharing distance of variable v. This measure lets us evaluate and compare NUMA effects on the algorithm. The average sharing distance for an algorithm is

$$\overline{S}_{dis} = \frac{\sum_{i=1}^{k} S_{dis}(v_i)}{k}.$$

Release cost (R_{cost}) measures the required overhead for an algorithm to release the lock and exit the CS. Generally, when a process exits the CS, that process informs another requesting process to enter the CS. The most efficient way to perform this is to inform only one process in the system. However, on a cache-coherent shared-memory system, the situation is more complex because all requesting processes might have to be informed, thus causing cache invalidations. There might also be some additional software overhead in algorithm executions, such as changing the spinning status of the requesting processes when a new process enters the CS. The release cost might degrade the performance of a mutual exclusion algorithm significantly.

Architecture and system effects

Parallel-computing performance on scalable shared-memory architectures depends mainly on the structure of the interconnection networks linking processors to memory modules, and on the efficiency of the memory- and cache-management systems. Different applications interact with these architectural factors in different ways. Therefore, designers of architectures and algorithms can benefit from a comparative performance evaluation that considers the architecture, the application algorithm, and the relationship between the two. Three major factors affect mutual exclusion algorithm performance in shared-memory systems: *memory hierarchies* from interconnection network structures and NUMA models, *cache effects* from NUMA models, and *hot spots* causing network and memory contention.

Memory hierarchies

The choice of interconnection networks to link processors to cache or memory modules can make NUMA times vary drastically, depending on the access patterns involved. Examples of interconnection networks for large-scale shared-memory multiprocessors are the multistage interconnection network (BBN Butterfly systems), hierarchical ring structure (KSR systems), cluster-based network (the Dash system), hierarchical bus (the Data-Diffusion Machine (DDM) system), and torus network (the Cray T3D system).

In large NUMA architectures, the memory-system organization also affects communication latency. NUMA memory systems fall into three types in terms of data migration and coherence:

- *Cache-coherent NUMA (CC-NUMA).* Each processor has an associated cache and a designated portion of the global shared memory. Cache coherence might be maintained by a directory-based cache-coherence protocol. The Dash system is an example.
- *Non-cache-coherent NUMA (Non-CC-NUMA).* This architecture either supports no local caches (for example, the BBN GP1000[2]), or provides a local cache that disallows caching of shared data, to avoid the cache-coherence problem (for example, the BBN TC2000[2]).
- *Cache-only memory architecture (COMA).* Like CC-NUMA, in COMA each processor has a cache and a designated portion of the global shared memory. However, COMA augments the memory of each processor to act as a large cache. A write-invalidate protocol maintains consistency among cache blocks in the system. A COMA system allows transparent migration and replication of data items to the processors where they are referenced. The KSR-1 is a COMA system.[3]

Cache effects

Caches let shared-memory systems effectively reduce average memory-access time by exploiting temporal and spatial localities. *Temporal locality* occurs in many programs: once a location (data or instruction) is referenced, it is often referenced again very soon. *Spatial locality* is the probability that once a location is referenced, a nearby location will be referenced soon. A local spin in a mutual exclusion algorithm demonstrates temporal locality. Data prefetching is another important cache mechanism, which generally exploits a program's spatial locality. Mutual exclusion algorithm implementations can effectively use prefetching to reduce the number of memory accesses.

However, *false sharing* and *cache pollution* are two side effects of cache prefetching. False sharing occurs when a cache miss causes the system to bring back a block of data containing more information than the required block. With an invalidation-based coherence protocol, false sharing will generate unnecessary invalidation

Table 1. Additional properties and complexities of several mutual exclusion algorithms.

Algorithm	FIFO	Time-bound	Contention sensitivity	# accesses
B-L	No	No	Moderate	$O(n)$
FIFO	Yes	No	Moderate	$O(n)$
Lamport	No	No	Moderate	$O(n)$
YA	Yes	No	No/1-sensitive	$O(\log_2(n))$
d-ary	No	No	Moderate	$O(\log_d(n))$

overheads. A mutual exclusion algorithm can eliminate false sharing by distributing shared variables well. Cache pollution means that additional data loaded in a cache for a cache miss are invalidated before they are referenced. Cache pollution is due to limited cache size and concurrent writes to the shared data. Cache pollution degrades system performance by consuming limited bandwidth between processors and memory modules. The additional invalidation time increases the average memory-access delay in a mutual exclusion algorithm.

Hot spots

Hot-spot contention on a network-based shared-memory architecture occurs when a large number of processors try to simultaneously access a globally shared variable across the network. A mutual exclusion operation generates a hot spot in a system. Hot-spot effects might degrade overall network traffic, not just the traffic to shared variables. However, hot-spot effects vary on different architectures.

A scalability metric for mutual exclusion algorithms

A comprehensive performance factor for evaluating and comparing mutual exclusion algorithms is scalability: how their performance on parallel machines increases as the number of processors increases. We have proposed a latency metric that uses network delay to evaluate parallel computing scalability.[4] Based on this concept, we use the following metric to measure and compare the scalabilities of mutual exclusion algorithms.

For a given mutual exclusion algorithm implementation on a given machine, let $L(N)$ be the average latency when the algorithm runs on N processors, and let $L(N')$ be the average latency when it runs on $N' > N$ processors. The average latency time is the process-response time. This is because we consider the total execution time of a mutual exclusion primitive to be an important latency source.

If the system size changes from N to N', the point-to-point scalability is

$$scale(N,N') = \frac{L(N)}{L(N')}.$$

In practice, this equation's value is less than or equal to 1. A large value means that the program and the architecture have small increments in latencies, providing efficient utilization of an increasing number of processors. On the other hand, a small value means large increments in latency. Because the processor increment in execution directly affects the scalability measurement, we use this increment as a weight variable. The weighted average scalability for a mutual exclusion algorithm running on a multiprocessor system up to p processors is

$$S = \frac{\sum_{i=2}^{p} 2^{(i-1)} \times scale(1,i)}{\sum_{i=2}^{p} 2^{(i-1)}}.$$

This metric concerns relative latency increments rather than absolute cycle times, so it can be used for scalability comparisons among different architectures.

Four representative mutual exclusion algorithms

To demonstrate our framework's usefulness, we chose these mutual exclusion algorithms:

- The *B-L algorithm*: an algorithm independently proposed by Burns and Lynch[5] and Lamport.[6]
- Lamport's algorithm without the time-bound requirement.[7]
- A FIFO-based algorithm proposed by Lycklama and Hadzilacos.[8]
- The *YA algorithm*: a tree-structured algorithm proposed by Yang and Anderson.[9]

Each algorithm maintains mutual exclusion, deadlock-freedom, and starvation-freedom. Table 1 lists their additional properties (the d-ary algorithm is a newly designed algorithm, which we will discuss later).

The B-L algorithm

Figure 2 outlines the B-L algorithm. In the first stage of this algorithm, each process sets its bit to indicate its request to enter the CS, and tests the bits of the lower-numbered processes. If it finds other CS requests, it gives up and restarts. On the other hand, if none of the lower-numbered processes has its bit set, the requesting process proceeds to the second stage, where it tests the bits of the higher-numbered processes, and waits for its turn.

```
Shared int x[N]; /* N is the number of
  threads */
Private int t;

Initial          ∀j::x[j]=0;

Process i;       /*i=0,1,...,N-1*/
 1. Do{
 2.              Non-CS;
 3. L:           x[i]=1;
 4.              for (t=0;t<i-1;t++)
 5.                if x[t]{
 6.                  x[i]=0;
 7.                  do {} while(x[t]);
 8.                  goto L;}
 9.              for(t=i+1;t<N;t++)
10.                do while (x[t]);
11.             CS;
12.             x[i]=0;}
13. while(1);
```

Figure 2. The B-L algorithm.

```
Shared int b[N],x,y; /* N is the number of
  threads */
private int j;

Initial      (∀j::b[j]==0)∧(y==-1);

Process i;   /*i=0,1,2,...,N-1*/
 1. Do{
 2.          Non-CS;
 3. L:       b[i]=1;
 4.          x=i;
 5.          if(y!=-1){
 6.            b[i]=0;
 7.            do {} while (y!=-1);
 8.            goto L;}
 9.          y=i;
10.          if(x!=i) {
11.            b[i]=0;
12.            for(j=0;j<N;j++)
13               do {} while (b[j]);
14.            if(y!=i){
15.              do {} while (y!=-1);
16.              goto L;}}
17.          CS;
18.          y=-1;
19.          b[i]=0;}
20. while(1)
```

Figure 3. Lamport's fast mutual exclusion algorithm.

Two loops spin on the elements of shared array $x[N]$, where N is the number of requesting processes. For element $x[i]$ $(0 \le i \le N-1)$, the maximum number of sharing degree $S_{deg}(x[i])$ is N. This indicates that the two spinning loops might generate hot variables.

The B-L algorithm is not contention-sensitive. When only one process contends for the CS, the process must visit $N-1$ shared variables. When process i exits from the CS and releases the CS, the processes spinning on $x[i]$ in statements 7 and 10 will be awakened. In addition, all the processes spinning in statement 7 will go back and restart after the CS is released. Therefore, all the additional operations executed because of a release of the CS are considered as the release cost (R_{cost}). However, the processes detained in the queuing loop (statement 9) will never restart. Thus, the more processes that can enter the queuing phase, the more efficient execution will be.

Lamport's algorithm

Figure 3 outlines Lamport's "fast mutual exclusion algorithm," which assumes that "contention for a critical section is rare in a well-designed system; most of the time, a process will be able to enter without having to wait."[7] This algorithm minimizes the number of memory accesses in the absence of contention. It uses two shared variables, x and y, for a requesting process to determine if contention for the CS exists. When the system has no contention, a process enters the CS by this statement sequence: $3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 17$.

When more than one process is contending for the CS, the first process executing statement 9 will set y to its identification so that statement 5 will prevent other processes from contending for the CS. For those processes that have passed statement 5, at most one process can enter the CS directly by passing statement 10, and the other processes turn to execute statements 11 and 12. The `for` loop in statement 12 guarantees that each contending process has set its $b[i]$ label to 0 before a process enters the CS by passing statement 14.

Lamport's algorithm has three spinning loops: statements 7, 13, and 15. Although these loops have the same average sharing degree (N in the worst case), the loops in statements 7 and 15 will generate higher contention than the loop in statement 13. This is because statements 7 and 15 always spin on the single shared variable y, while statement 13 spins on different elements of array b. This algorithm has a higher number of spinning operations than the B-L algorithm.

Because this algorithm provides two possible execution paths to the CS for a process, the probability for each path to be executed determines the average process-response time. When a high number of processes are contending to enter the CS, the probability of a process entering the CS by satisfying $(x!=i)$ in statement 10 is small. In most cases, it enters the CS by satisfying $(y!=i)$ in statement 14. Further, when a high number of

```
Shared int c[N],v[N],turn[N][2];
   /* N is the number of threads */
Private int j,bit,s[N][2];

Initial ∀j::(v[j]==0∧c[j]==0);

Process i; /*i=0,1,...,N-1*/
 1. Do{
 2.    Non-CS;
 3.    c[i]=1;
 4.    for (j=0;j<N-1;j++){
 5.      s[j][0]=turn[j][0];s[j][1]=
           turn[j][1];}
 6.    bit=1-bit;
 7.    turn[i][bit]=1-turn[i][bit];
 8.    v[i]=1;
 9.    c[i]=0;
10.    for(j=0;j<N;j++)
11.      do{}while(c[j]∨(v[j]∧s[j][0]==
             turn[j][0]∧s[j][1]==
               turn[j][1]));
12.    Entry(i);
13.    CS;
14.    Exit(i);
15.    v[i]=0;}
16. while(1);
```

Figure 4. A FIFO-based algorithm.

processes contend for the CS, at most two can enter the CS consecutively: one enters by passing statement 10, another enters by passing statement 14, and the rest of them go back and restart. This shows that Lamport's algorithm has a potentially high release cost.

The FIFO-based algorithm

To study the cost of a fairness requirement on mutual exclusion algorithms, we selected a FIFO-based algorithm (see Figure 4).[8] In this algorithm, a specific segment of the program, called the *doorway*, enacts the FIFO property. In Figure 4, `Entry(i)` and `Exit(i)` are the entry code and the exit code. The doorway comprises statements 3 to 10, which are surrounded by a variable $c[i]$ that is set to 1 on entry to the doorway and to 0 after exiting the doorway. Moreover, in the doorway code, each process owns a two-bit communication variable $turn[i]$ ($turn[i][0]$ and $turn[i][1]$). The algorithm modifies this variable's value in each iteration of the process by alternately complementing the value of one of its bits.

When a process wants to enter the CS, it first saves all the other communication variables in statements 4 and 5. It then changes its own communication variable in statements 6 and 7, signals its intention to enter the interior by setting variable $v[i]$ to 1 in statement 8, exits the doorway in statement 9, and waits to contend for the CS in statements 10 and 11. The number of shared-variable accesses for a process to execute the doorway code is $2N + 3$.

```
Shared int c[N][N],p[N][N],t[N][N];
   /* N is the number of threads */
Private int rival,j,k,l,high,comp;

Initial ∀i,j:0 ≤ j < log2(N)::c[j,i] =
   -1 ∧ p[j,i] = 0;

Process i: /*i=0,1,...,N-1*/
 1. j=0;k=i/2;l=i;
 2. high=(int)(log(N)/log(2));
 3. if(log(NPROCESS)/log(2)>high)
 4.   high++;
 5. Do {
 6.   Non-CS;
 7.   while(j<high){
 8.     c[j][l]=i;
 9.     t[j][k]=i;
10.     p[j][i]=0;
11.     if(l%2==0) comp=l+1;
12.       else comp=l-1;
13.     rival=c[j][comp];
14.     if(rival!=-1){
15.       if(p[j][k]==i){
16.          if(p[j][rival]==0)
               p[j][rival]=1;
17.          do while(p[j][i]==0);
18.          if(t[j][k]==i)
19.             do while(p[j][i]<=1);
20.       }
21.     }
22.     k=k/2;l=l/2;j++;
23.   }
24.   CS;
25.   while(j>0){
26.     k=i/power(2,j+2);l=i/power
          (2,j+1);j--;
27.     c[j][l]=-1;
28.     rival=t[j][k];
29.     if(rival!=i) p[j][rival]=2;}
30.   }
31. while(1);
```

Figure 5. The YA algorithm, which is tree-based.

The YA algorithm

This algorithm (see Figure 5) constructs a binary tree-based structure. To enter the CS, a requesting process must traverse a path from the leaves up to the root, executing the entry section code along the path. Upon exiting the CS, the process traverses this path in reverse, executing the exit section code.

Along the path for a process to enter the CS, two spinning loops are at statements 17 and 18. Each statement's sharing degree is 2 because the spinning variable $p[j][i]$ is accessed by two concurrent processes at most. No process in this algorithm has to restart after a release of the CS. This feature eliminates the release cost. Moreover, each process has a unique execution path from its entry to the exit. The process-response time is $O(\log(N))$. The algorithm is not contention-sensitive.

Yang and Anderson have proposed another algorithm that combines Lamport's fast mutual exclusion algorithm

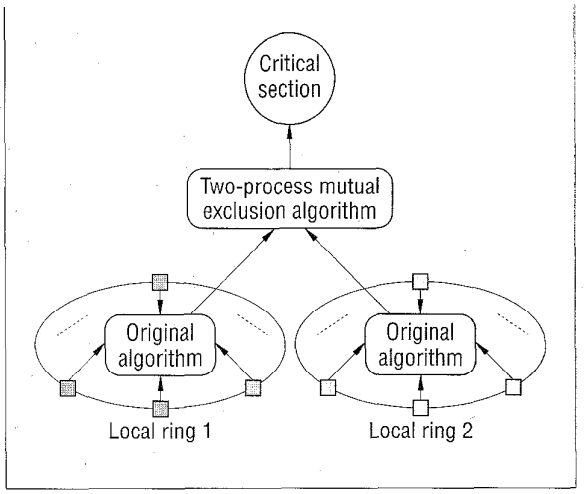

Figure 6. The two-level refined mapping structure for mutual exclusion algorithms on the KSR-1.

and their tree-based mutual exclusion algorithm.[9] This algorithm's process-response time is $O(N)$. This algorithm is *1-sensitive*—that is, it performs best if only one process contends for the CS.

Performance evaluation on the TC2000 and the KSR-1

The BBN TC2000 supports up to 512 Motorola 88100 processors, each operating at 20 MHz.[2] Its network uses a butterfly switch composed of 8×8 switches.

The KSR-1 is a hierarchical-ring-based system with up to 1,088 64-bit custom superscalar RISC processors (20 MHz).[3] A basic ring unit in the KSR-1 has 32 processors. The system uses a two-level hierarchy to interconnect 34 rings (1,088 processors). Each processor has a 32-Mbyte cache.

The basic structure of the KSR-1 is the slotted ring, which divides the ring bandwidth into a number of message slots circulating continuously through the ring. The number of slots is equal to the number of processors plus the number of directory/router cells connecting to the upper (level-1) ring. A standard KSR-1 ring has 34 slots: 32 for the processors and two for the cells. Each slot can be loaded with a packet consisting of a 16-byte header and 128 bytes of data. This packet, called a *subpage*, is the KSR-1's basic data unit. A processor on the ring that is ready to transmit a message waits until an available empty slot rotates through that processor's ring interface.

We performed our experiments on a TC2000 with 64 processors and on a two-ring KSR-1 with 64 processors. We measured the average time to acquire and release the lock for different numbers of processors. Each processor requested lock acquisition 1,000 times in a loop. Two additional instructions set the CSs for these experiments to make them nonempty and to check each execution's correctness. We also ran all experiments under benchmark mode. Therefore, the systems were solely used for these measurements.

Architecture effects

The following architecture effects can cause differences in the performance of mutual exclusion algorithms on the TC2000 and the KSR-1:

NUMA effects

In both systems, a processor accesses shared-memory and cache space at different distances with different timing costs. However, the NUMA models of the two systems differ.

The TC2000 has only two levels of memory accesses—local and remote—between which the difference in access time is approximately a factor of 4.7. The average access time for read/writes is 0.42 μs for local and 1.96 μs for remote.

On the KSR-1, a memory access can have multiple distances: the subcache, the local cache, a remote cache in the local ring (remote), and a remote cache in a remote ring (remote-remote). The access-time difference can be up to about 28.5 times, not considering subcache access. The average access time for read/writes is 0.1 μs for subcache, 2.5 μs for local, 9.5 μs for remote, and 28.5 μs for remote-remote.

Non-CC-NUMA versus COMA

The TC2000 avoids the cache-coherence problem by disallowing caching of shared data, while the KSR-1 is a cache-coherence and cache-only system. Because of this difference in memory systems, a mutual exclusion algorithm will exhibit different execution patterns on each machine.

System reactions to hot spots

The TC2000 is much more hot-spot-sensitive than the KSR-1, because of different network structures. Our previous research indicates that ring-network transactions in the KSR-1 decrease no more than 50% in the presence of memory hot spots.[10] This compares with a 300% latency change in the TC2000.

Process sequences

The KSR-1's rotating slotted-ring network can naturally order CS requests to be sequenced by their locations on a ring. Although this ordering property does not provide a FIFO service, it guarantees that any request-

ing process will eventually execute its CS. However, a software structure for FIFO service might cause additional overhead because of rearrangement of the natural ring orders.

Table 2. Sharing degree (S_{deg}) and sharing distance (S_{dis}) of the naive and refined mutual exclusion algorithms.

	Naive (on TC2000 and KSR-1)		Refined (on KSR-1)	
	S_{deg}	S_{dis}	S_{deg}	S_{dis}
B-L	n	global	$n/2$	local
FIFO	n	global	$n/2$	local
Lamport	n	global	$n/2$	local
YA	2	global	2	local
d-ary	n/a	n/a	d	local

Locality issues

The KSR-1 system is more locality-sensitive than the TC2000 because of its hierarchical structure.

Results

The KSR-1's architecture and memory systems are complex, and the ways of implementing a software-based algorithm on it can affect the algorithm's performance significantly. To show this, we implemented the mutual exclusion algorithms on the KSR-1 in two different ways:

- *Naive mapping:* We fully used the shared-memory illusion provided by the KSR-1 for easy programming, without considering detailed structures of the ring network and the COMA system.
- *Refined mapping:* We varied the algorithm implementations to account for the effects of the architecture and memory system. (We did not refine the FIFO algorithm because its major component is the refined B-L algorithm.)

The TC2000's architecture and memory systems are less complex. So, we used only a naive mapping on the TC2000.

When refining each base algorithm to improve its performance on the KSR-1, we had three main concerns. First, program-thread scheduling and execution should efficiently exploit hierarchical locality by processing referenced data in a local cache or in caches in the local ring as much as possible. Second, the algorithm designs should consider the structure of the COMA system to minimize cache-access misses and cache invalidations. This consideration includes the effective use of the cache subpage, which will bring a group of variables to a local cache to reduce remote accesses. Finally, we must explicitly manage processor locality by manually mapping the data structures among the rings without using the compiler and system's mapping options.

To exploit the system's localities, a refined implementation maps the original algorithm into two local rings. The process in each local ring selects a leader. A two-process management in the global ring coordinates the two leaders contending for the CS. Figure 6 shows this two-level refined mapping structure, which reduces the average sharing degree and the average sharing distance.

Table 2 compares the sharing degree and sharing distance for the naive and refined algorithms on the TC2000 and the KSR-1.

Process-response times

Figure 7a presents process-response times for the mutual exclusion algorithms on the KSR-1. For naive mapping, the FIFO algorithm's process-response time sharply increased for more than 48 processors because of the overhead of forming the FIFO queue. Lamport's fast algorithm also performed poorly because of high release costs. The B-L algorithm and YA algorithm performed the best among the four.

The refined mapping improved performance significantly. The refined B-L algorithm performed better than the refined YA algorithm, for three reasons. First, the B-L algorithm's first stage rules out a large number of requesting processes. This filtering effectively reduces the number of restarted threads. Second, the B-L algorithm does not force the requesting processes to enter the CS in a certain pattern, such as a tree. So, the B-L algorithm should have less software delay time in practice, although it might have higher contention than the YA algorithm. However, the KSR-1's ring structure might effectively handle network contention.[10] Finally, in the B-L algorithm, the shared variables are vectors. This structure lets us use the cache subpage (128 bytes), which will bring up to 32 integer variables at a time to a local cache for reduction of remote accesses.

Figure 7b presents process-response time on the TC2000. The results show that the YA algorithm can be more effectively implemented on the TC2000 than on the KSR-1, for two main reasons.

First, the TC2000's multistage interconnection network (MIN)-based architecture is much more hot-spot sensitive than the KSR-1.[10] Except for the YA algorithm, all the algorithms are hot-spot sensitive. Hot-spot effects can decrease significantly on the TC2000 if shared variables in an algorithm are widely distributed or efficient delays are used to reduce the number of

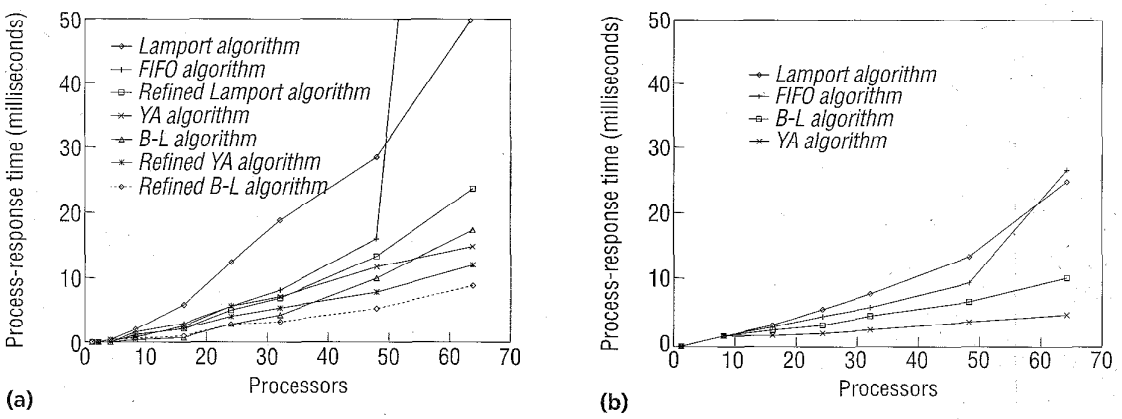

Figure 7. Process-response times for the mutual exclusion algorithms: (a) on the KSR-1, (b) on the TC2000.

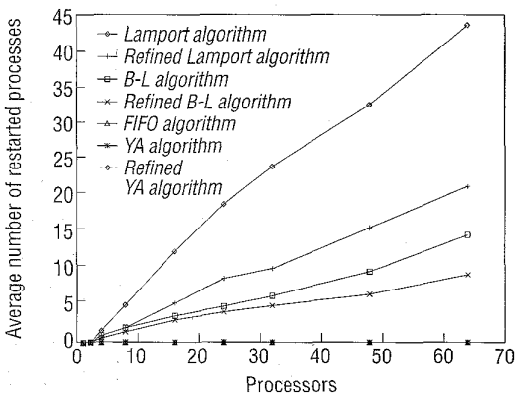

Figure 8. Release cost for the mutual exclusion algorithms and their refined versions on the KSR-1.

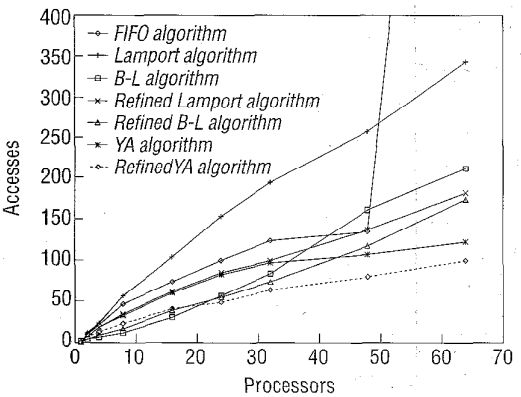

Figure 9. Remote memory accesses for the mutual exclusion algorithms and their refined versions on the KSR-1.

remote accesses to the shared variables. For example, the YA algorithm uses a tree structure to distribute shared-variable accesses.

Second, the TC2000's simple local/remote memory-access model generates straightforward overhead patterns. In contrast, dynamic data migrations and cache coherence on the KSR-1 complicate algorithm mapping, and can seriously degrade performance if the COMA system is not properly used.

Release costs

We measure the release cost by counting the average number of restarted processes caused by each release of the CS in an algorithm. In this study, we conducted experiments only on the KSR-1 because the release cost is mainly algorithm-dependent, rather than architecture-dependent.

Figure 8 presents release costs. The YA algorithm and its refined version have no release costs. The FIFO algorithm has an insignificant number of restarted threads because the additional FIFO software filter generates a limited number of waiting processes for the CS. The results indicate that the refined implementations reduce the release costs for the Lamport fast algorithm and the B-L algorithm. This is because the refined implementations exploit more localities of the ring architecture and reduce network contention. Therefore, the number of waiting processes decreases, which reduces the number of restarted threads.

Remote accesses

Besides being the only complexity measure in standard theoretical analysis, an algorithm's number of remote accesses is important to execution evaluation.

Figure 9 reports the number of remote accesses for each algorithm and its refined version on the KSR-1 as the number of processors increases to 64. The number of remote accesses for each algorithm's execution is consistent with that algorithm's theoretical complexity analysis. The YA algorithm has the fewest remote accesses, followed by the B-L, Lamport, and FIFO algorithms. Also, the number of remote memory accesses for each refined implementation is significantly lower than the number of accesses for the corresponding naive version. This further shows the effectiveness of the refined implementations.

Fairness

As our results have shown, a mutual exclusion algorithm with FIFO support can have tremendous overhead caused by execution of the additional software structure. We ran a group of fairness tests on the KSR-1 to observe whether the competing processes in a non-FIFO algorithm can still fairly access the CS in any given period of execution time. We evaluated fairness by measuring the variance of the numbers of accesses to the CS from each processor for each algorithm.

Figure 10 presents the standard deviation results. The variance of the access sequence is relatively insignificant. For example, the highest standard deviation measure of the B-L algorithm is 0.35. Therefore, additional software might not be necessary to support the FIFO sequence, unless the application requires it.

Contention sensitivity

We compared the contention sensitivity of the four algorithms and their refined versions on up to 64 processors of the KSR-1, using these conditions:

- When the total number of processes is less than eight, all processes contend for the CS. Thus, when the process number increases from one to eight, the contention to the CS increases, which will degrade performance of both contention-sensitive algorithms and noncontention-sensitive algorithms.
- When the total number of processes is more than eight, only eight random processes contend for the CS; all the other processes do the work without accessing the CS. The contention to the CS stays unchanged. Thus, the performance of contention-sensitive algorithms should also stay unchanged.
- All the processes in an application are randomly distributed over the hierarchical ring structure.

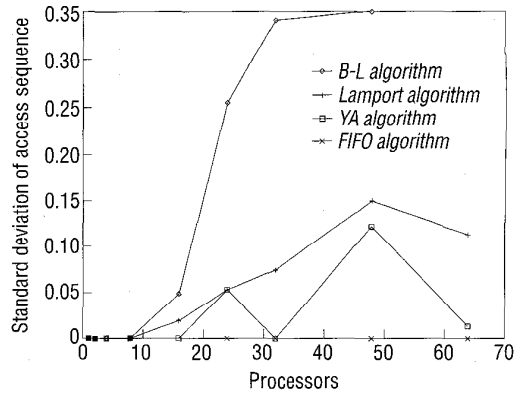

Figure 10. Fairness of the mutual exclusion algorithms on the KSR-1.

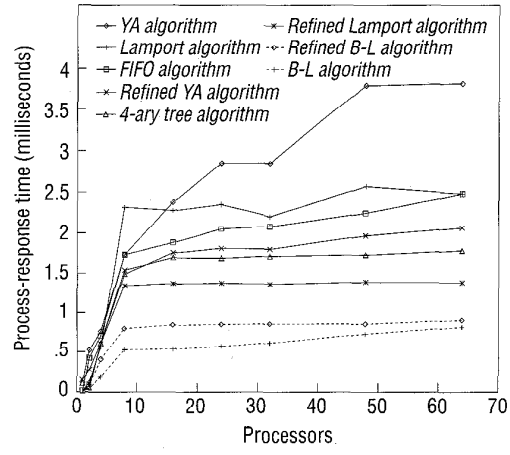

Figure 11. Process-response times for the mutual exclusion algorithms and their refined versions on the KSR-1.

Figure 11 shows that the B-L, Lamport, and FIFO algorithms have more stable contention sensitivity than does the YA algorithm. Except for the YA algorithm, shared variables/vectors form the basic data structures of the algorithms. The response time of these algorithms depends only on the number of processes contending for the CS. Therefore, after more than eight processors are used, the response time of the three algorithms remains roughly constant.

The YA algorithm uses a binary tree. The total number of processes determine the tree's height and width. That is the main reason why this algorithm's contention sensitivity differs from that of the other algorithms. However, the refined YA algorithm's contention sensitivity becomes more stable, because the new implementation reduces network contention. Testing contention sensitivity can also reveal the performance

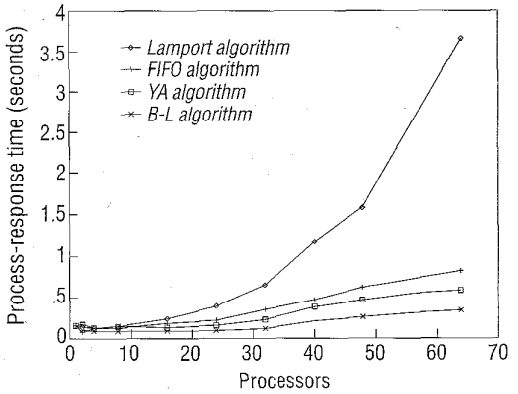

Figure 12. Execution times for implementation of the mutual exclusion algorithms in a parallel quicksort algorithm on the KSR-1.

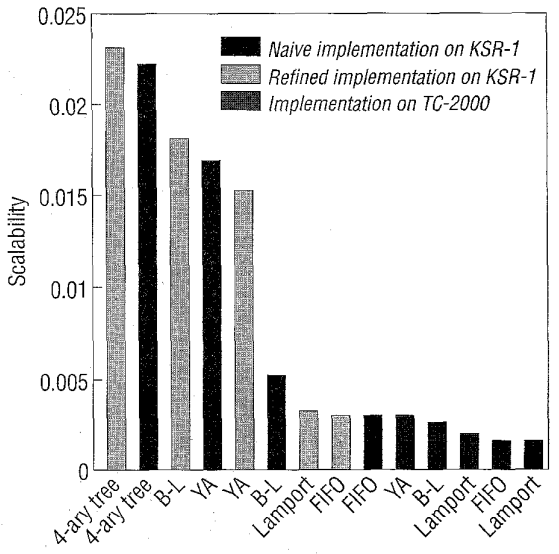

Figure 13. Scalability of the mutual exclusion algorithms on the KSR-1 and the TC2000.

changes of the algorithms with different CS-access patterns in an application program.

Performance in an application program

We implemented the mutual exclusion algorithms in a parallel quicksort (QS) algorithm to evaluate their effects on the program. In this algorithm, a queue stores and allocates tasks. The task queue is a shared resource. Each process visits the task queue exclusively when it generates a task and places it into the queue, or when it requests a new task from the queue. The efficiency of mutual exclusion in the QS algorithm significantly affects the algorithm's execution performance.

We measured the process-response time of each mutual exclusion algorithm implemented in the QS algorithm on the KSR-1 (see Figure 12). The Lamport algorithm performed the slowest. As we expected, the B-L and YA algorithms performed significantly better. These results are consistent with the previous performance results for these algorithms.

Scalability

Finally, we compared the scalability of the algorithms on both machines. We used the scalability metric to compare the algorithms' access latency in increments from two to 64 processors. The measurements quantitatively give the scaling characteristics of each algorithm on each machine (see Figure 13).

An interesting result concerns relative-latency increments. Absolute access-delay times for some algorithms on the KSR-1 are similar to those on the TC2000 because of the faster ring network. However, the relative-latency increment for each algorithm is lower on the KSR-1 because of cache coherence and fewer hot-spot effects.

The measurements also show that algorithm scalability is architecture-dependent. For example, the YA algorithm on the KSR is more scalable than the ones on the TC2000. These results further confirm our analysis of the architectural effects on, and implementation variations of, these algorithms.

Three new efficient algorithms

We'll now describe the structure and performance of the three fast mutual exclusion algorithms we've developed. We have previously prepared correctness proofs of each algorithm.[11]

The *d*-ary tree algorithm

Based on our analyses and experiments, an efficient mutual exclusion algorithm in a shared-memory system should have these characteristics:

1. It should use hierarchical data structures to exploit architectural locality and reduce hot-spot effects.
2. It should use a vector to allocate shared variables to exploit cache-prefetching for possibly enhancing its contention sensitivity, and to reduce the number of remote-memory accesses.
3. Because shared-memory architectures vary, its structures must be adjustable to adaptively optimize its performance on different machines.

Furthermore, our qualitative and quantitative results indicate that the binary tree structure in the YA algorithm can effectively distribute network contention and eliminate the release cost (Characteristic 1). However, the algorithm is not contention-sensitive. On the other hand, the B-L algorithm effectively uses the cache-prefetching function and is more contention-sensitive (Characteristic 2). It responded faster than the YA algorithm on the KSR-1.

The *d-ary tree algorithm* is a hybrid that combines the good features of the YA and the B-L algorithms. d is an integer (>1) that can adaptively change to adjust algorithm performance on different architectures (Characteristic 3). (Table 1 lists this algorithm's additional properties, and Table 2 lists its sharing degree and sharing distance on the TC2000 and the KSR-1.)

Figure 14 outlines the algorithm, which constructs a logical d-ary tree as distributed processes in the system. Entry and exit sections are associated with each link in the tree. The two code sections are similar to the B-L algorithm. To enter the CS, a requesting process must traverse a path from its leaves up to the root, executing the entry section of each link along the path. Upon exiting the CS, the process traverses the path in reverse, executing each link's exit section.

The tree structure is similar to that of the YA algorithm. However, the tree structure's adjustable parameter makes this algorithm unique and adaptive. When d increases, the tree's height decreases, which means the traveling distance of accessing the CS decreases, but contention and the release cost might also increase. When d is larger than the number of processes in the system, the d-ary tree algorithm is equivalent to the B-L algorithm. When $d = 2$, the algorithm constructs a binary tree, but it differs from the YA algorithm, because the contending process is based on the B-L algorithm.

The d-ary tree algorithm satisfies the basic requirement of mutual exclusion. (It is a straightforward proof based on the correctness of the B-L algorithm and the tree structure. The formal proof is beyond the scope of this article.) This algorithm does not guarantee FIFO order. The spinning loops of statements 11 and 15 in Figure 14 have the same sharing degree, d. Although the algorithm's performance still depends on the total number of processes, the adaptive tree structure makes it more contention-sensitive than the YA algorithm, because it takes advantage of cache prefetching.

We measured the process-response times of the d-ary tree algorithm running on the KSR-1 (see Figure 15a) and the TC2000 (see Figure 15b) for different values of d. On the KSR-1, the 4-ary algorithm performed best. Compared with the fastest B-L algorithm the 4-ary algorithm's process-response time decreased approximately 40%. On the TC2000, the 2-ary algorithm performed best, close to the YA algorithm.

```
Adjustable parameter: d /* the number of
   children of a node */

Shared int x[N][N]; /* N is the number of
   threads */
Private int high,k,i,j,l,len;

Initial ∀j,i::x[j][i]=0;

Process id; /*id=0,1,...,N-1*/
 1. j=0;k=id/d;l=id;
 2. len=NPROCESS;
      /*len records the number of nodes at
      level j*/
 3. high=logd(NPROCESS);
 4. Do {
 5.   Non-CS;
 6.   while (j<high){
 7.   yy: x[j][l]=1;
 8.     for (i=k*d;i<l;i++)
 9.       if (x[j][i]){
10.         x[j][l]=0;
11.         do {} while (x[j][i]);
12.         goto yy;
13.       }
14.     for (i=l+1;i<min((k+1)*d,len);i++)
15.       do {} while (x[j][i]);
16.     l=l/d;k=k/d;j++;
17. len=(len%d==0)?len/d:len/d+1;}
18. CS;
19. j=high;
20. pow2=power(d,high)
21. while (j>0){
22.   j--;
23.   pow2/=d;
24.   l=id/pow2;
25.   x[j][l]=0;
26.   }
27. }
28. while(1);
```

Figure 14. The d-ary tree algorithm.

A FAST 2-ARY TREE ALGORITHM

As we have shown qualitatively and quantitatively, a non-cache-coherent, hot-spot-sensitive architecture, such as the TC2000, might favor execution of a low-order tree algorithm, while a cache-coherent architecture, such as

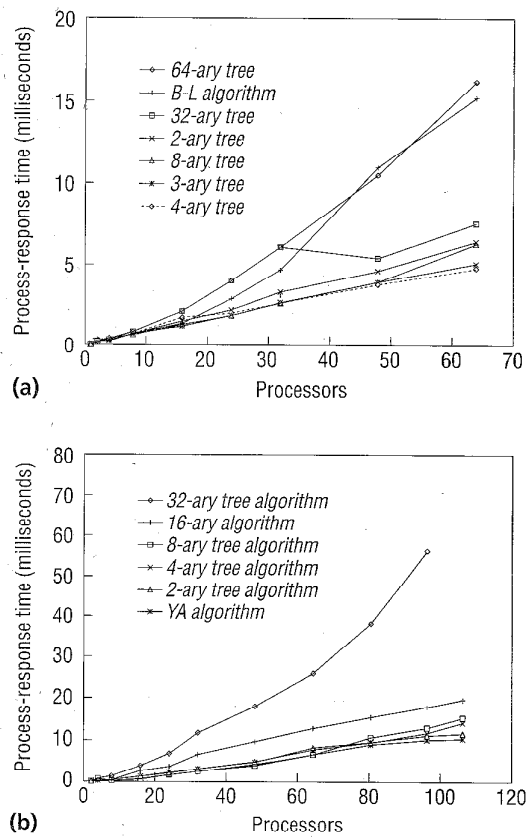

Figure 15. The process-response times of the *d*-ary algorithm for different values of *d*: (a) on the KSR-1, (b) on the TC2000.

the KSR-1, might favor execution of a moderate-order tree algorithm. We propose a variation of the 2-ary tree algorithm that includes local spins. Because this algorithm outperforms all the YA and the 2-ary tree algorithms on the TC2000 and the KSR-1, we call it the *fast 2-ary tree algorithm*.

This algorithm's construction is based on a two-process mutual exclusion algorithm (see Figure 16). Our algorithm associates each process i with only two variables: $x[i]$ and $c[i]$. Allocation $c[i]$ is the local place on which process i spins. Element $x[i]$ informs the rival of process i of its status of contending for the CS and of its identity number (i). This algorithm's basic data structure has a low memory-allocation requirement compared with other algorithms of the type. For example, in the original 2-ary tree algorithm,[9] each process needs three variables, and an additional variable resolves the contention.

In practice, a mutual exclusion algorithm might not require fairness. The fast 2-ary tree algorithm resolves process-request contention by favoring one of the two contention processes, which eliminates any delay requirement. In contrast, the original 2-ary tree algorithm uses a dynamic contending method to resolve the contention, which tends to generate extra delay overheads.

In the algorithm in Figure 16, process u has higher priority than process v, while both contend for the CS simultaneously. When process u executes, it sets its identifying number u in $x[0]$ to announce its involvement and sets $c[u]$ to 1 to prepare for resolving con-

```
Shared int x[2],c[N];

Initialization: ∀ i ∈ [0,N-1]::c[i]= 1; x[0]=x[1]=-1;

Process u;                                  Process v;
int rival;                                  int rival;
Do {                                        Do {
u1: Non-CS;                                 v1:  Non-CS;
u2: x[0]=u; (R)                             v2:  x[1]=v; (R)
u3: c[u]=1;                                 v3:  yy: if((rival=x[0])!=-1) (R)
u4: if ((rival=x[1])!=-1){ (R)              v4:    c[rival]=0; (R)
u5:   c[rival]=0; (R)                       v5:    do {} while(c[v]);
u6:   do {} while(c[u]);}                   v6:    c[v]=1;
u7: CS;                                     v7:    goto yy;}
u8: x[0]=-1; (R)                            v8:  CS;
u9: if(rival!=-1) (R)                       v9:  x[1]=-1; (R)
while(1).                                   v10: c[rival]=0;} (R)
                                            while(1).
```

Figure 16. The two-process mutual exclusion algorithm for the fast 2-ary tree algorithm.

```
Shared int x[N][N], c[N][N];

Initialization: ∀i,j ∈ [0,N-
  1]::(c[i][j]=1or0)∧(x[i][j]=-1);

Process id; /*id=0,1,...,N-1*/
int high,i,j,l,rival;
 1. j=0; l=id;
 2. high=⌈log2(NPROCESS)⌉;
 3. Do {
 4.   Non-CS;
 5.   while (j<high){
 6.     if(l%2==0){
 7.       x[j][l]=id;
 8.     c[j][id]=1;
 9.       if ((rival=x[j][l + 1])!=-1){
10.         c[j][rival] = 0;
11.         do {} while(c[j][id]);
12.         }
13.      }
14. else{
15.   x[j][l]=id;
16.   yy: if ((rival=x[j][l-1])!=-1){
17.     c[j][rival=0;
18.     do {} while (c[j][id]);
19.     c[j][id]=1;
20.     goto yy;
21.     }
22.   }
23.   l/=2;j++;
24. }
25. CS;
26. j=high;
27. pow2= power(d,high)
28. while (j>0){
29.    j--;
30.    pow2/=d;
31.    l=id/pow2;
32.    x[j][l]=-1;
33.    temp=(l%2==0)?l+1:l-1;
34.    if((rival=x[j][temp])!=-1)
35.      c[rival]=0;}
36. }
37. while(1);
```

Figure 17. The fast 2-ary tree algorithm.

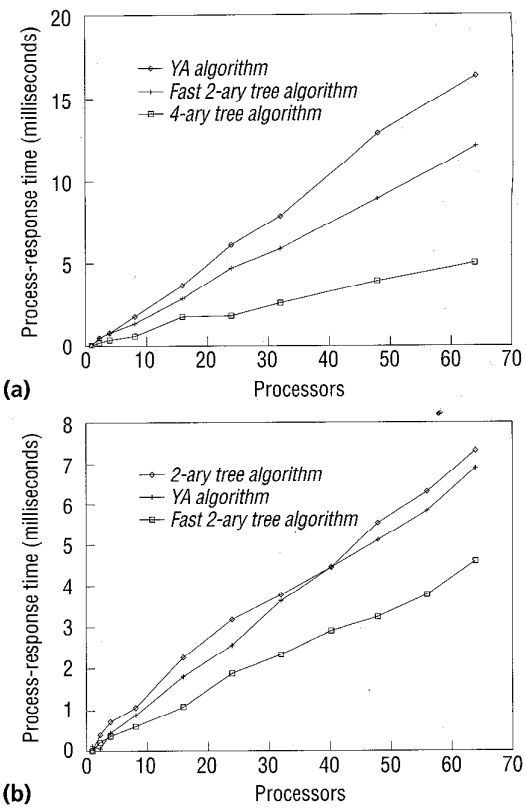

Figure 18. Process-response times of the fast 2-ary tree algorithm: (a) on the KSR-1, (b) on the TC2000.

tention. When process u detects its rival's ambition to enter the CS in statement $u4$, it cleans the rival's executing path in $u5$ to guarantee that its rival will let it pass the busy waiting loop in $u6$ later. Process v can enter the CS only when process u finishes contending for the CS.

Regarding the algorithm's memory-access complexity, process u has only four remote-memory accesses in the presence of contention, and five in the absence of contention. Process v will execute four remote accesses in the absence of contention. In an average situation, processes u and v execute the CS in turns; process v only needs to test in statement $v3$ twice in the presence of contention. Thus, process v has an average of seven remote-memory accesses (in Figure 16, remote accesses are marked with (R)). So, for this two-process mutual exclusion algorithm, the average number of remote accesses per process is (4 + 4)/2 (= 4) in the absence of contention, and (5 + 7)/2 (= 6) in the presence of contention. In contrast, for the YA algorithm, the average number of remote memory accesses per process is five in the absence of contention and 10 in the presence of contention.

Figure 17 shows the fast 2-ary tree mutual exclusion algorithm. To avoid interference among node mutual exclusion algorithms in the tree, each node employs an independent set of variables. In each node algorithm, the process with an even value of l has priority over the process with an odd value of l when contending for the CS. Because the rival of a process changes dynamically, the process must check whether its rival still exists when it exits the CS (see statement 34 in Figure 17).

We compared the performance of the fast 2-ary tree algorithm, the YA algorithm, and the best ones of the d-ary tree family on the KSR-1 (see Figure 18a) and TC2000 (see Figure 18b). The fast 2-ary tree algorithm outperforms the YA algorithm on both machines, which is consistent with our analyses.

```
Shared int busy[N],wait[N],turn; /* N is
  total number of processes */
Shared int gate1,gate2;

Initialization: (∀i ∈ [0,N-1]::(busy[i]=0;
wait[i]=1))∧(turn=N;gate1=gate2=0);

Process id: /*id=0,1,...,N-1*/
int I, temp=0;
 1. Do {
 2.   Non-CS;
 3.   busy[id]=1;
 4.   do {} while(gate1||gate2);
 5.   if(turn!=N){do{}while(wait[id]);
        wait[id]++;}
 6.   else {
 7.     gate2=1;
 8.     for(i=0;i<id;i++)
 9.       if(busy[i]){
10.         temp=1;
11.         goto L;}
12. L: if(temp){
13.   gate2=0; temp=0;
14.   do {} while(wait[id]); wait[id]++;}
15. else
16.   {turn=id; gate2=0;}
17. }
18. CS;
19. gate1=1; i=(turn+1)%N;
20. while(i!=turn&&busy[i]==0) i=(i+1)%N;
21. if(i!=turn){
22.   turn=i;
23.   busy[id]=0;
24.   wait[turn]=0;}
25. else{
26.   busy[id]=0; wait[id]=1;
27.   turn=N;}
28. gate1=0;
29. }.
```

Figure 19. A fast algorithm for a high-contention environment.

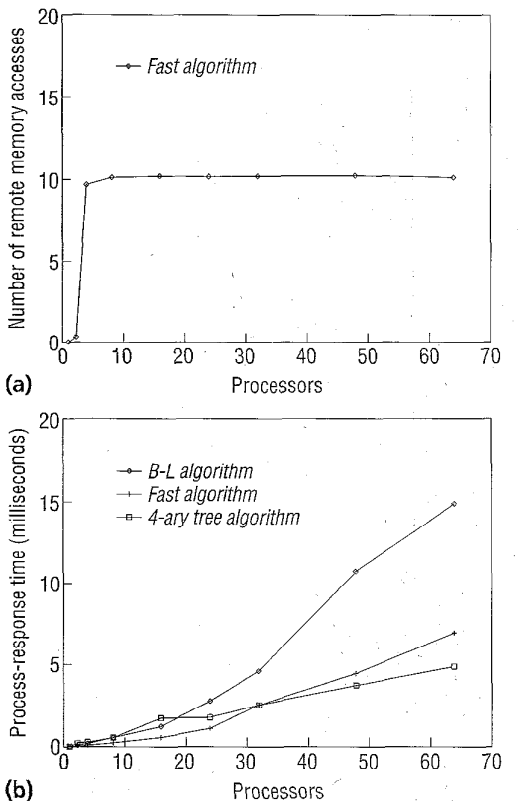

Figure 20. Performance of the fast algorithm on the KSR-1: (a) memory-access complexity, (b) process-response time.

A FAST ALGORITHM FOR HIGH-CONTENTION ENVIRONMENTS ON CACHE-COHERENCE SYSTEMS

Lamport proposes a fast algorithm for the noncontention environment, which covers a special class of applications.[7] On the other hand, another special class of applications frequently accesses the CS and generates high contention. We've developed a fast algorithm for this class of applications on cache-coherence shared-memory systems.

Let W denote the average number of processes waiting for access to the CS. In a high-contention environment, W is usually greater than 1, which means that when a process tries to access the CS, another process always executes the CS. In this case, a fast path directing the new processes to spin locally would be effective. When a process releases the CS, many waiting processes immediately want to enter the CS. So, it would be efficient to directly transfer ownership of the CS to one of the waiting processes.

Figure 19 shows our algorithm. It allocates each process i in two shared vector variables: *busy*[i] and *wait*[i]. *busy*[i] reports whether process i is contending for the CS. The algorithm allocates this vector in as few cache blocks as possible to take advantage of cache prefetching. *wait*[i] is the spinning variable of process i, which is local to process i.

Another important shared variable is *turn*, which reports whether the CS is occupied. If so, *turn* records the process ID number in the CS. All the processes are queued into a logical ring, according to their process ID numbers: $p0 \rightarrow p1 \rightarrow p2 \cdots \rightarrow p(N-1) \rightarrow p0$. *gate1* and *gate2* are two gate variables: *gate2* is used only in a non-contention situation to protect the selection of the initial

owner of the CS; *gate1* blocks a process from entering the spinning state when the CS is released.

In a high-contention environment, when a process begins contending for the CS, it usually enters the spinning state at statement 5 by one write and three reads. If a process finds that *turn* is equal to N at statement 5, it enters the selection process of the initial owner of the CS at statement 7. Then it either enters the spinning state at statement 14 or becomes the initial owner to enter the CS.

The memory-access complexity of the selection process is $O(N)$. When a process exits its execution in the CS, it searches the logical ring to transfer the CS to the nearest waiting process. So, the memory-access complexity for release is $O(N/W)$. In a high-contention environment, W approaches N, and thus the memory-access complexity decreases to $O(1)$. Hence, the average complexity of this algorithm in a high-contention situation can be as good as $O(1)$, which is independent of the total number of processes.

To further confirm this, we measured our fast algorithm on the KSR-1 in a high-contention environment where all the processes repeatedly request entrance to the CS. The fast algorithm has an almost flat memory-access curve (see Figure 20a), which is consistent with the predicted constant memory-access complexity. Its process-response time (see Figure 20b) is shorter than that of the B-L algorithm, but slightly longer than that of the 4-ary tree algorithm. In addition, this fast algorithm has no release cost.

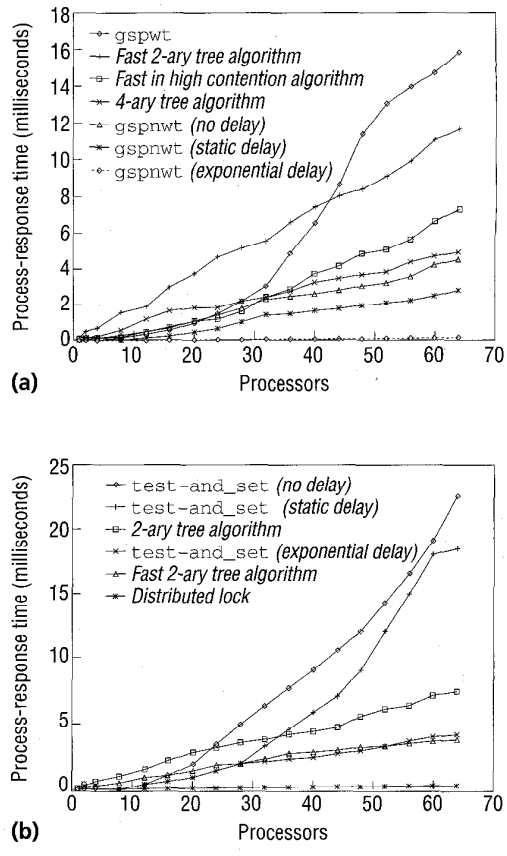

Figure 21. The process-response times of our software-based algorithms and the hardware primitives: (a) on the KSR-1, and (b) on the TC2000.

Comparing the new algorithms with hardware primitives

To show the efficiency of our software-based algorithms, we compared their performance with that of efficient hardware primitives on the KSR-1 and the TC2000.[1] The primitives we selected from the KSR-1 are the `gspwt` (get subpage wait), and the `gspnwt` (get subpage no wait) and its variations: access the lock with no delay, access the lock with a static delay, and access the lock with an exponential delay. These primitives are highly effective on the KSR-1. The primitives we selected on the TC2000 are the `test_and_set` and its variations: access the lock with no delay, access the lock with a static delay, and access the lock with an exponential delay. For comparison, we also used the performance of a distributed lock. We compared the process-response times in a high-contention environment.

Figure 21a shows the process-response time on the KSR-1. Our algorithms outperformed `gspwt` and rivaled the `gspnwt` variations. More specifically, the 4-ary algorithm nearly outperformed the `gspnwt` with no delay. This is important because `gspnwt` and `gspwt` depend on the KSR-1's architecture. These results indicate that software-based algorithms could be viable alternatives.

Figure 21b shows the process-response time of the same algorithms on the TC2000. Our algorithms outperformed or rivaled all variations of the `test_and_set` algorithms. The process-response time curve of the best software algorithm, the fast 2-ary tree algorithm, is reasonably close to that of the distributed lock.

Our results show that the execution performance of both hardware-based and software-based algorithms is architecture-dependent. Generally, a hardware-based algorithm performs more efficiently than a software-based algorithm in the presence of contention, because of support from atomic instructions. However, we can use the flexibility of software-based algorithms to take advantage of the structure of architectures and systems. The performance of the proposed algorithms shows their potential to rival hardware-based algorithms.

Based on our experimental research, we believe that software mutual exclusion algorithm studies directed by theoretical memory-access complexities probably are reaching a theoretical limit. For example, the maximum number of remote memory accesses of a mutual exclusion algorithm would not be lower than $O(\log(n))$, where n is the number of remote memory accesses. Algorithms with this complexity have been developed (for example, the YA and d-ary algorithms). On the other hand, approaches that combine analyses and experiments could significantly improve the performance of the algorithms.

Acknowledgment

We are grateful to the anonymous referees for carefully reading this article and for their constructive comments and suggestions. This work has been supported in part by the National Science Foundation under grants CCR-9102854 and CCR-9400719, by the US Air Force under research agreement FD-204092-64157, by the Air Force Office of Scientific Research under grant AFOSR-95-01-0215, and by a Fellowship from the Southwestern Bell Foundation. We conducted part of the experiments on the BBN TC2000 at Lawrence Livermore National Laboratory, and on the KSR-1 machines at Cornell University and at the University of Washington.

References

1. X. Zhang, R. Castañeda, and E.W. Chan, "Spin-Lock Synchronization on the Butterfly and KSR-1," *IEEE Parallel & Distributed Technology*, Vol. 2, No. 1, Spring 1994, pp. 51–63.
2. *Inside the GP1000 and the TC2000*, BBN Advanced Systems and Technologies, Boston, 1989.
3. *KSR-1 Technology Background*, Kendall Square Research, Cambridge, Mass., 1992.
4. X. Zhang, Y. Yan, and K. He, "Latency Metric: An Experimental Method for Measuring and Evaluating Program and Architecture Scalability," *J. Parallel and Distributed Computing*, Vol. 22, No. 3, Sept. 1994, pp. 392–410.
5. J. Burns and N. Lynch, "Mutual Exclusion Using Indivisible Reads and Writes," *Proc. 18th Ann. Allerton Conf. Communications, Control, and Computing*, Allerton House, Monticello, Ill., 1989, pp. 833–842.
6. L. Lamport, "On Inter-Process Communication—Parts I and II," *Distributed Computing*, Vol. 1, No. 2, 1986, pp. 77–101.
7. L. Lamport, "A Fast Mutual Exclusion Algorithm," *ACM Trans. Computer Systems*, Vol. 5, No. 1, 1987, pp. 1–11.
8. E.A. Lycklama and V. Hadzilacos, "A First-Come-First-Served Mutual Exclusion Algorithm with Small Communication Variables," *ACM Trans. Programming Languages and Systems*, Vol. 13, No. 4, Oct. 1991, pp. 558–576.
9. J.-H. Yang and J. Anderson, "Fast, Scalable Synchronization with Minimal Hardware Support," *Proc. 12th Ann. ACM Symp. Principles of Distributed Computing*, ACM Press, New York, 1993, pp. 171–182.
10. X. Zhang, Y. Yan, and R. Castañeda, "Comparative Performance Evaluation of Hot Spot Contention Between MIN-Based and Ring-Based Shared-Memory Architectures," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 8, Aug. 1995, pp. 872–886.
11. Y. Yan and X. Zhang, "Designing Fast Software Mutual Exclusion Algorithms on NUMA Systems," tech. report, High Performance Computing and Software Laboratory, Univ. of Texas at San Antonio, 1994.

Xiaodong Zhang is an associate professor of computer science and director of the High-Performance Computing and Software Laboratory at the University of Texas at San Antonio. His research interests are parallel and distributed computation, parallel architecture and system performance evaluation, and scientific computing. He received his BS in electrical engineering from Beijing Polytechnic University, China, in 1982 and his MS and PhD in computer science from the University of Colorado at Boulder in 1985 and 1989. He can be contacted at the High-Performance Computing and Software Laboratory, Univ. of Texas, San Antonio, TX 78249; zhang@ringer.cs.utsa.edu.

Yong Yan is a PhD candidate in computer science at the University of Texas at San Antonio. His research interests are parallel and distributed computing, performance evaluation, operating systems, and algorithm analysis. He was a visiting scholar in the High-Performance Computing and Software Laboratory at UTSA from 1993 to 1995. He received his BS and MS in computer science from Huazhong University in 1984 and 1987. He can be contacted at the High-Performance Computing and Software Laboratory, Univ. of Texas, San Antonio, TX 78249; yyan@dragon.cs.utsa.edu.

Robert Castañeda is a PhD student in computer science at the University of Texas at San Antonio. His research interest is in performance evaluation of parallel and distributed architectures and systems. He previously was a research associate at the High-Performance Computing and Software Laboratory at USTA. He received his BS and MS in computer science from UTSA in 1990 and 1994, and won the 1994 University Life Award for academic performance. He can be contacted at the High-Performance Computing and Software Laboratory, Univ. of Texas, San Antonio, TX 78249; rcastane@dragon.cs.utsa.edu.