A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors

Shiwa S. Fu, *Member, IEEE Computer Society*, and Nian-Feng Tzeng, *Senior Member, IEEE*

Abstract—Mutual exclusion in shared-memory multiprocessors is realized by employing a lock to determine the processor among those which compete for the critical section. Accesses to such a mutual exclusion lock may create heavy synchronization traffic and/or serious contention over the network, thereby degrading system performance considerably. In this paper, we introduce an efficient scheme which keeps synchronization traffic low and avoids serious hot-spot contention. This is made possible by constructing a circular list of the processors waiting for the critical section and by dispersing accesses to the lock. Extensive simulation of the proposed approach was conducted and the lower bound on the elapsed time was derived. Our simulation results demonstrate that the proposed scheme indeed achieves better performance than prior techniques, with its elapsed time close to the lower bound for the whole range of simulated system sizes, thus promising good scalability for large systems.

Index Terms—Circular lists, critical sections, hot-spot contention, linked lists, multiprocessors, mutual exclusion, tree of locks.

1 Introduction

Mutual exclusion with respect to a particular data structure requires that no more than one processor hold that data structure at a time. A lock is commonly used to realize mutual exclusion, guaranteeing that at most one processor at a time proceeds to the critical section.

A large multiprocessor system, such as NEC Cenju-3 [1] and Cedar [2], contains many processors and memory modules interconnected by the multistage interconnection network (MIN). To implement mutual exclusion in such a multiprocessor usually results in many processors acquiring a common variable (i.e., lock) located in a remote memory module. As a result, it not only causes serious contention at the remote memory module containing the lock, called a *hot spot*, but also crowds the network paths from that memory module all the way back to processors, known as *tree saturation*, which blocks other normal network traffic and thereby demotes system performance. This phenomenon lasts until every involved processor accomplishes its critical section. It is shown in [3] that even a small percentage of requests from each processor destined for the particular memory module can severely degrade the effective communication bandwidth of a large system.

Some early researchers in this area resorted to hardware approaches [3], [9], [14] and others sought software solutions [10], [11]. Hardware approaches [3] incorporate certain hardware in the interconnection network to trap and combine access requests heading toward the same memory location for hot-spot relief. However, the cost overhead due to added hardware poses a major concern. A less costly hardware combining technique has been introduced [9]. Goodman et al. [14] proposed the use of a synchronization bit, Queue_On_Lock bit, for implementing mutual exclusion. Their scheme requires the support of cache-coherence to maintain a linked-list queue, where the header processor of the queue becomes the owner of the lock and the queue members spin on their local cache lines, waiting to be awakened.

Instead of seeking a hardware solution, Anderson [10] considered a software approach, called the array-based queuing lock, where each processor uses the fetch_and_increment atomic instruction to get a unique location on which it spins. This scheme successfully disperses processors waiting for a single spin location at a remote memory module and lets each waiting processor spin on a location situated at a different remote memory module, with all spin locations formed an array.

Mellor-Crummey and Scott's [11] list-based queuing lock (called the MCS lock) was inspired by the Queue_On_Lock bit described in [14], but implemented in software. This scheme uses a fetch_and_store atomic instruction on a lock located in a remote memory module to link the involved processors to form a single waiting list. The header of the linked list acquires the lock, while any other processor, polling the lock later than the header, appends itself to the list and then goes to spin on a local memory location until it is awakened. On completion, the processor with the lock writes a wake-up message into the next processor's spin location, passing the lock. Their experimental result shows that the MCS lock has a better performance than Anderson's array-based queuing lock. The reason is that each member processor in MCS's linked list spins on a local memory location without going through the network, while each waiting processor in Anderson's array queue spins on

• *The authors are with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.*
E-mail: tzeng@cacs.usl.edu

Manuscript received 5 Sept. 1995.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number 101294.0.

a designated flag located in a different remote memory module. Spinning on a remote flag must go through the network, thereby taking much longer time and consuming network bandwidth, when compared with spinning on a local memory location.

Both Anderson's array-based queuing lock and MCS's list-based queuing lock are efficient software approaches for mutual exclusion in the multiprocessor system. However, there are still some drawbacks associated with these schemes:

1) Both of them require all involved processors to compete for a single lock located at a remote memory module in realizing mutual exclusion, rendering the memory module containing the lock a hot spot. The accesses to this single lock tend to create extremely heavy traffic contention.
2) In Anderson's scheme, each processor wait-spinning on a different remote flag must traverse the network, thereby consuming lots of network bandwidth and interfering with other processors' accesses to memory modules. MCS's linked list remedies this drawback by spinning on local memory locations. However, the MCS scheme suffers from one problem caused by the process of forming a linked list, as stated in the following. Each processor first issues a fetch_and_store instruction to the remote lock to obtain its predecessor's address (if any). It then sends a message containing its own address to notify its predecessor of reversing the link direction, referred to as *linked list redirection*. The linked list redirection operation must go through the network, consuming appreciable network bandwidth.

It should be noted that each waiting processor in Anderson's array queue spins at least once on the remote flag to acquire the lock (actually, waiting processors require more spins as their ranks in the queue increase), while each waiting processor in MCS's linked list issues only one linked list redirection operation. So, Anderson's scheme tends to generate more synchronization traffic through the network, exhibiting inferior performance.

As a result, a scheme able to overcome the above drawbacks is highly desired for large MIN-based multiprocessors. An efficient lock scheme is proposed in Section 2. In Section 3, we compare four mutual exclusion schemes. A lower bound on a parameter of interest is derived in Section 4. Our experimental results are demonstrated in Section 5. Concluding remarks, including a discussion about the implementation of different schemes on bus-based multiprocessors and future work, are provided in Section 6.

2 Proposed Approach

Our approach is briefly outlined first, followed by its detailed description in Section 2.1 and Section 2.2.

Instead of employing one lock located in a remote memory module, we use a tree of locks, assigning each lock to a different remote memory module. For a tree of locks, we define locks in the leaf level as *leaf locks*, the lock at the root as the *root lock*, and locks in between as *interior locks*. Furthermore, any list formed by linearly linking processors which share a leaf lock is called a *local list*. A *local circular list* (called *local CL* for short) is a local list with the last element pointing to the first element. An *intermediate CL* (or list) and the *global CL* (or list) are defined with respect to an interior lock and the root lock in a similar manner. The header of the global CL (or list) is the *global header*.

The purpose of our scheme is to form a global CL through these locks. The global CL, like MCS's linked list or Anderson's array queue, decides the sequence order for involved processors to enter the critical section, ensuring mutual exclusion. To form a global CL, all processors involved in mutual exclusion use the fetch_and_store atomic instruction to compete for the locks (in the tree). The process of forming a global CL starts with leaf locks to build local CLs, and then constructs intermediate CLs by merging local CLs through interior locks, until the global CL is formed using the root lock.

A $\lceil \log_n N \rceil$-level tree of locks with degree n is used in our scheme for a MIN-based multiprocessor with N processors involved in mutual exclusion, which are divided into $\lceil N/n \rceil$ groups. To avoid possible memory contention, it is preferable that all nodes in a lock tree reside in separate memory modules [15]. For a system with N memory modules, the largest lock tree is a tree with the minimum fanout, i.e., $n = 2$, and the total number of nodes in such a tree is $\frac{N}{2} + \frac{N}{4} + \ldots + 2 + 1 = N - 1$. Thus, it is always possible to spread these locks across N separate memory modules. As an example, a system with $N = 8$ and $n = 2$ is shown in Fig. 1, where circles and squares are employed to represent the locks and the processors, respectively. The symbol beside each lock indicates in which memory module a lock resides.

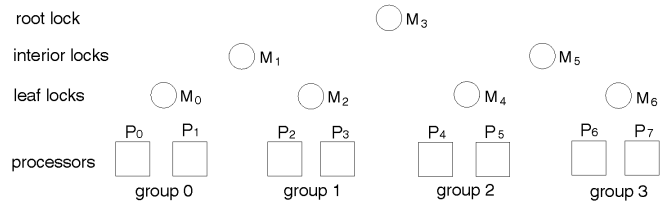

Fig. 1. A system with $N = 8$ processors divided into four groups with $n = 2$ processors each, requiring a three-level tree of locks with degree two.

In a lock tree, the *parent lock* of a particular lock is defined as the lock in the next higher level (toward the root lock) of the same subtree. For example, the lock in M_1 in Fig. 1 is the parent lock of the locks in M_0 and in M_2. Similarly, the parent lock of the locks in M_1 and M_5 is resided in M_3. Initially, a leaf lock is assigned to each group. Processors in each group compete for their assigned common leaf lock to build a local CL. Totally, $\lceil N/n \rceil$ local CLs are created in the system. For each CL, only the header competes for its parent lock in an interior level (or the root level if the tree has only two levels), while members halt, waiting to be awakened. Again, these $\lceil N/n \rceil$ headers are partitioned into $\lceil N/n^2 \rceil$ groups, with n headers in a group sharing one common parent lock in an interior level. The n

headers in a group compete for the common lock to form a single intermediate CL by merging their corresponding n CLs, where each of these CLs contains $n-1$ members and one header. As a result, the total number of intermediate CLs after merging is $\lceil N/n^2 \rceil$, with n^2-1 members and one header in each intermediate CL. This merger process repeats progressively until all the intermediate CLs constitute one global CL (at the root level). This global CL contains $N-1$ members and one global header.

It should be noted that the above discussion considers only a specific situation where all processors in the same group are assumed to form one single CL. However, processors in one group in general may form multiple CLs, depending on when they issue the synchronization requests. The single lock assigned to a group guarantees forming one CL at a time; when a CL is completed, it serves to form another CL. The header of each CL, knowing that it is the header, participates in merging. Our subsequent discussion deals with general situations where processors in the same group could form an arbitrary number of CLs, one at a time.

Our scheme has the attributes of

1) selecting one processor to enter the critical section as soon as possible, and
2) passing the lock to the next processor (by notifying it) immediately after a processor is done with the critical section, as long as there is any waiting processor.

The first attribute results from global lock acquisition described below, while the second attribute is due to privilege consignation stated in Section 2.2.

2.2 Global Lock Acquisition

Every processor wishing to enter the critical section executes the global lock acquisition procedure, as demonstrated by a C-like code in Fig. 2, to select one processor to enter the critical section as soon as possible. In this code, array L keeps the assigned lock locations along the path from a leaf to the root in the lock tree. Specifically, L[1] keeps the memory location where the assigned leaf lock resides, L[root_level] has the location for the root lock, and L[i], $2 \le i \le$ root_ level - 1, is for the assigned intermediate locks at level *i*. As an example, L[i], $1 \le i \le 3$, for processor P_0 in Fig. 1 contains the lock locations, respectively, in memory modules M_0, M_1, and M_3, while it is M_4, M_5, and M_3 for processor P_4. Variable I contains two fields: **wait** and **next**; I->wait is a Boolean flag indicating the spin status of the current processor, while I->next records the address of the next processor in the CL (or list). Initially, I->wait and I->next are set respectively to a FALSE and to the current processor's own address (i.e., processor id). All locks in the system are initialized with nil.

In Fig. 2, the first iteration of the loop is to form a local CL for each group, iteration$_2$ till iteration$_{\text{root_level-1}}$ merge local CLs (or intermediate CLs) in the same group into an intermediate CL, and finally a global list is formed through the last iteration. Within each loop, the first fetch_and_store atomic instruction is responsible for forming a linked list, and this linked list is closed into a CL through the second fetch_and_store instruction. The details of how it works are stated in the following.

Every processor in the same group issues the first fetch_and_store to fetch a value from, and store its I->next value into, the assigned lock. Any processor which acquires a nil (i.e., I->next == nil) from the assigned lock becomes the header; otherwise, it becomes a list member. Once a processor knows itself as a header, it then issues the second fetch_and_store to acquire the next processor's address (for closing the linked list into a CL), and, meanwhile, resets the assigned lock to nil. On the other hand, when a processor knows itself as the list member, it exits the loop and halts, waiting to be awakened by spinning on its local flag. Once a processor becomes a header after its first fetch_and_store instruction, any processor in the

```
#define root_level = ⌈log_n N⌉;
struct node {
    boolean wait;
    struct node *next;
};
typedef struct node lock[1..root_level];
Acquire_lock(lock L, node *I) {
    int level;
    I->wait = FALSE;                /* reset spin flag */
    I->next = my_proc_id;           /* set 'next' as myself */
    for (level = 1; level ≤ root_level; level++) {      /* a global list will be formed after this loop */
        I->next = fetch_and_store(L[level], I->next);   /* form a list */
        if (I->next != nil) {       /* I am a member of the CL */
            I->wait = TRUE;         /* set spin flag */
            break;                  /* exit loop */
        }
        if (level < root)           /* only enforce on leaf lock and intermediate locks */
            I->next = fetch_and_store(L[level], nil);   /* form a CL */
    }
    while(I->wait);     /* member processor spins local flag, waiting for being awakened */
}
```

Fig. 2. Global lock acquisition of the proposed scheme.

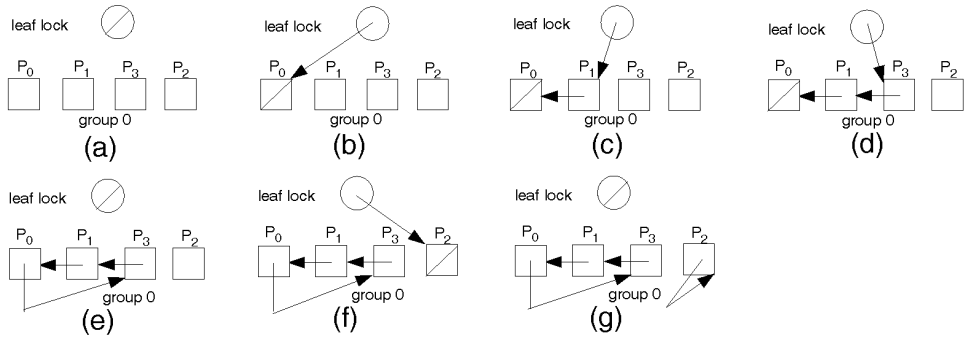

Fig. 3. Two local CLs built in the same group through a leaf lock.

same group may be added to the linked list as a member before the header closes it into a CL. Actually, each processor fetches the address of its next processor (in the list) from the lock and adds itself to the list through the fetch_and_store instruction. Note that our approach does not require linked list redirection, as in MCS's mentioned earlier. Processors arrive at the critical section temporally differently. If a processor arrives at the critical section late and fetches the lock right after the lock is reset by the header's second fetch_and_store instruction, it becomes the header of a separate CL.

The above idea is better illustrated by an example in Fig. 3, where a slash symbol in a circle represents the lock containing a nil value, a slash symbol in a square is a header, and arrow (node *i*) → (node *j*) indicates that node *i* contains node *j*'s address.

Once a CL is formed (after issuing the second fetch_and_store), the header (i.e., the processor which acquires a nil from the assigned lock) starts to access its parent lock in the next iteration, attempting to merge its corresponding CL with other CLs into an intermediate CL. This merger process repeats progressively until a global linked list is constructed at the root level. The global header then has the privilege to enter the critical section.

Fig. 4 shows a system with N = 16 processors divided into four groups with n = 4, requiring a two-level tree of locks. Only the root lock is depicted. Initially, it is assumed that two separate local CLs were formed each in group 0 and group 2, and one local CL in group 1 and in group 3, as shown in Fig. 4a. Fig. 4b and Fig. 4c illustrate that two local CLs in group 0 are merged into a global linked list. Processor P_0 is the global header, implying that P_0 actually acquires the lock and is allowed to enter the critical section. Now, the global header does not issue the second fetch_and_store instruction to close the global linked list, so as to permit as many upcoming CLs as possible to be merged in the current global linked list during the period when the global header is executing its critical section.

2.2 Privilege Consignation

After the global header finishes with the critical section, it consigns the lock privilege to the next processor, if any, in the global list. Here, we propose a new atomic instruction, swap_and_compare, which is a modification to the previous compare_and_swap instruction used in the MCS scheme [11]. The semantics of these two instructions are contrasted in Fig. 5, where our proposed instruction first exchanges the content of the variable "old" with that in "L," then assigns "new" to "L" if "L" equals "old."

The pseudocode for our privilege consignation (or lock release) is illustrated in Fig. 6, with each line numbered to facilitate explanation of this code in the following. For a

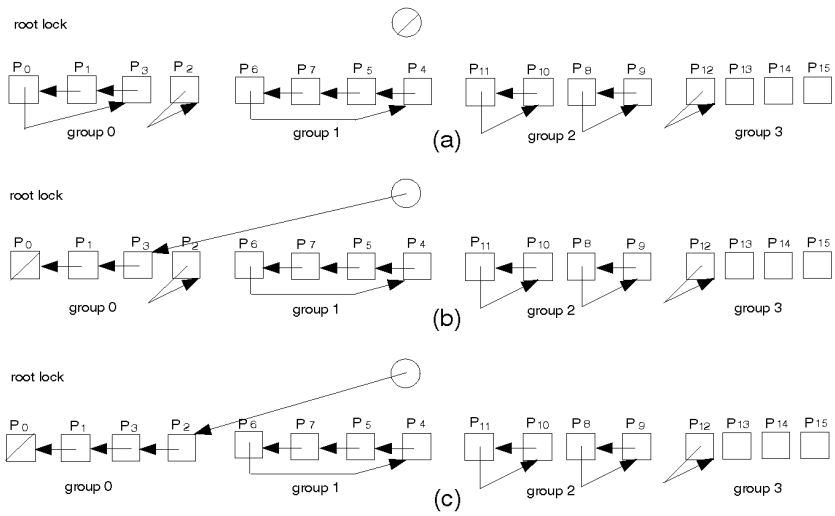

Fig. 4. Two CLs in group 0 merged into a global linked list through the root lock.

```
compare_and_swap (L, old, new) {
    if (L == old) {
        L = new;
        return TRUE;
    }
    else
        return FALSE;
}
```

```
swap_and_compare (L, old, new) {
    swap(L, old);
    if (L == old)
        L = new;
}
```

Fig. 5. compare_and_swap and swap_and_compare primitives.

```
0  Release_lock(lock L, node *I) {
1      if (I->next == nil) {                  /* I am a global header */
2          while (1) {                        /* repeatedly consign privilege until releasing lock */
3              I->next = my_proc_id;          /* set 'next' as myself */
4              swap_and_compare (L[level], I->next, nil);    /* try to release lock */
5              if (I->next == my_proc_id)     /* no processor is waiting for privilege */
6                  return;                    /* release lock successfully */
7              I->next->wait = FALSE;         /* wake up next processor */
8              I->wait = TRUE;                /* set spin flag */
9              while(I->wait);                /* spin local flag, waiting for being awakened */
10         }
11     }
12     I->next->wait = FALSE;                 /* wake up next processor */
13 }
```

Fig. 6. Privilege consignation of the proposed scheme.

member processor in the global linked list, the process of privilege consignation is simple: it just wakes the next processor up by setting its spin flag to a FALSE (see line 12 in Fig. 6). As for the global header (whose I->next == nil), it issues a swap_and_compare atomic instruction toward the root lock, resulting in the following two possible scenarios depending on the retrieved value from the root lock.

1) If the retrieved value (saved in I->next) is the global header's own address, no processor except the global header itself is in the global list, implying that the lock is released successfully (see lines 4–6 in Fig. 6);
2) If the retrieved value is not the global header's own address, the global list is closed into a global CL with multiple members in the CL; the global header, therefore, consigns the privilege to the next processor in the global CL, and then halts, waiting to be awakened (see lines 4 and 7–9).

To better explain the above scenarios, an example follows.

As mentioned earlier, multiple separate local and intermediate CLs are likely to be formed during the process of global lock acquisition, which determines the global header (or the first processor say, P^f denoted by the shaded square in Fig. 7) to enter the critical section. However, no more than one global CL may exist at any time to guarantee mutual exclusion. To do this, P^f issues a swap_and_compare instruction to store its address (rather than a nil) in the root lock, as illustrated by Fig. 7a. With this, all subsequent headers (of intermediate CLs), if any, would form a global list when accessing the root lock before the last member of the global CL finishes with the critical section, as given by Fig. 7b and Fig. 7c. As soon as the last global CL member exits from the critical section, it wakes up P^f (i.e., the first processor that entered the critical section), which then makes the existing global list become the only global CL by accessing the root lock once (using a swap_and_compare instruction), as depicted in Fig. 7d. Now, the privilege of entering the critical section is again consigned to every member in the newly formed global CL in sequence, as before. In the meantime, all headers (of intermediate CLs) which access the root lock, if any, would constitute a global list until members in the current global CL are all exhausted. This process repeats till no more processors are waiting for the privilege, causing P^f eventually to release the lock as shown in Fig. 7e.

The privilege consignation procedure in Fig. 6 relies on the swap_and_compare instruction to release the lock. Basically, the swap_and_compare instruction executes two operations atomically:

1) exchanges the content of the root lock with that of the I->next, and
2) resets the root lock if two swapped variables have the same content.

Now, considering the case that only the fetch_and_store atomic instruction is available and no swap_and_compare primitive is supported. The global header in this case must issue two separate fetch_and_store instructions to realize the above two operations in a non-atomic manner, resulting in more complicated lock release as depicted in Fig. 8, where the first two fetch_and_store instructions are to accomplish the above two operations. A member processor in this case, however, behaves exactly the same as that in Fig. 6.

Privilege consignation from one competing processor to another depends on whether or not any header (of an intermediate CL) accesses the root lock during the course of (1) privilege consignation among members of the global CL, and (2) resetting the root lock (by P^f). As soon as all members of the current global CL are exhausted, P^f is awakened (by the last CL member when it exits from the critical section) and accesses the root lock using the fetch_and_store instruction, as demonstrated in Fig. 8. If no other header

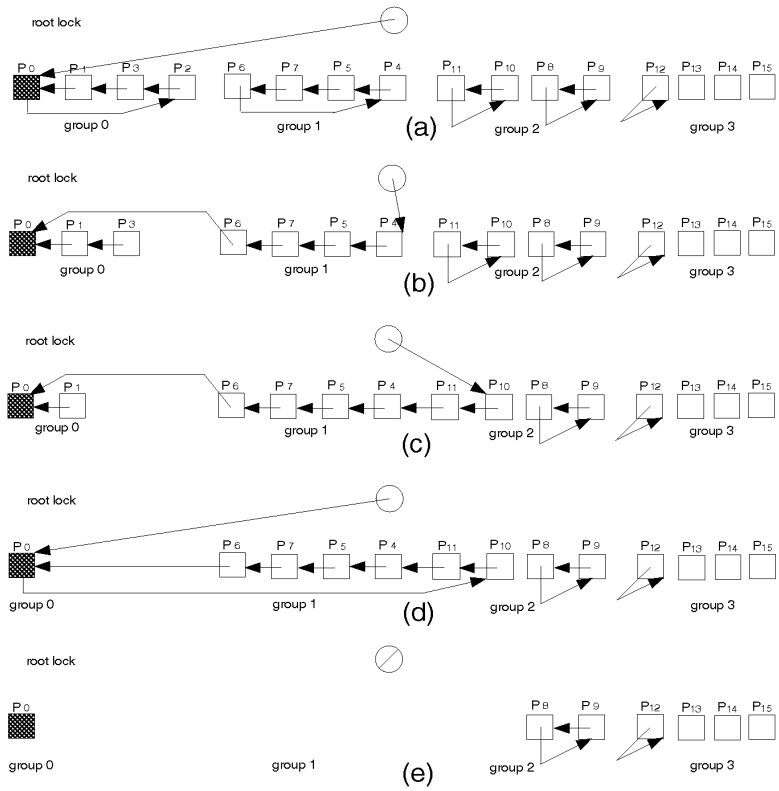

Fig. 7. One global CL and one global linked list coexisting to realize fast privilege consignation.

accesses the root lock during the course of situation 1 above, as shown in Fig. 9a, no other intermediate CL is waiting for the privilege and thus P^f proceeds to reset (i.e., store a nil in) the root lock; otherwise, a global linked list exists and one (or more) intermediate CL(s) is (are) waiting for the privilege, requiring the member(s) in the existing list to be granted the privilege one by one, as indicated by state 1 of Fig. 8, which corresponds to lines 2-4 and 7-10 in Fig. 6.

If no header accesses the root lock during the course of situation (2) above, P^f releases lock successfully, as denoted by state 2 of Fig. 8, which corresponds to lines 3-6 in Fig. 6. On the other hand, if at least one header accesses the root lock, as illustrated in Fig. 9b, the root lock value fetched through the reset fetch_and_store instruction has to be stored back to the root lock (called *restoration*) using another fetch_and_store instruction. A global CL formed by the reset fetch_and_store instruction is shown in Fig. 9c. Note that the situation of Fig. 9c is similar to that of group 0 in Fig. 4a, which happens during the course of global lock acquisition.

Again, if no header accesses the root lock during the course of restoration by P^f, no more intermediate CL is

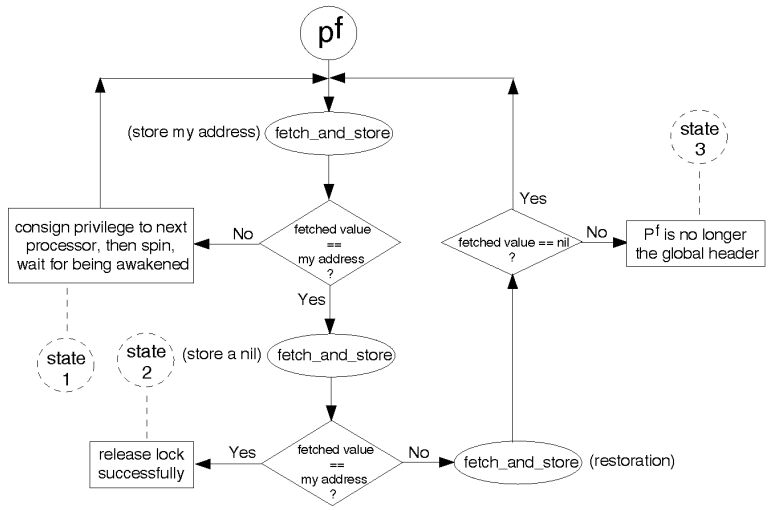

Fig. 8. Privilege consignation of global header without the swap_and_compare primitive.

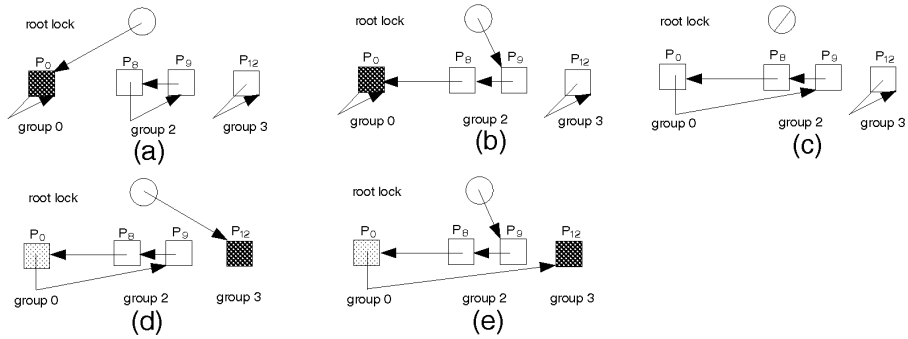

Fig. 9. Privilege consignation leading to state 3.

waiting, leading back to the start point in Fig. 8 (with its situation similar to that of group 0 in Fig. 4b). However, if at least one header accesses the root lock during restoration, a separate global list is formed, as shown in Fig. 9d where the list consists of P_{12}. Now, this restoration makes the global CL merged with the newly formed global list, yielding a single global linked list as illustrated in Fig. 9e. As a result, P^f is no longer the global header (as shown by the gray square) and P_{12} becomes the new global header due to acquiring a nil from the root lock as shown in Fig. 9d, leading to state 3 in Fig. 8. Under this circumstance, P_0 halts, waiting to be awakened. When P_0, a gray square, is awakened later on, it consigns the privilege to its next processor. In the next section, we discuss hot spot contention associated with the previous schemes and how it is solved by our proposed scheme through reducing the number of accesses to the root lock.

3 Comparison of Mutual Exclusion Schemes

Typical mutual exclusion in a multiprocessor system consists of three activities: acquire lock, wait lock, and release lock. For all previous schemes, processors acquiring lock must compete for a single lock located at a remote memory module, possibly rendering the memory module containing the lock a hot spot. As only one processor can get the lock at a time (according to mutual exclusion), the other competing processors must wait for the lock to come; they usually spin on a flag located in local memory or remote memory. Spinning on remote memory must go through the interconnection network, creating considerable traffic over the network. Release lock usually takes a negligible time because it just consigns the lock privilege to the next processor (if any) or resets the lock, without confronting any competitors. On the other hand, acquire lock and wait lock take much longer times, especially in the presence of a hot spot resulting from acquire lock in the previous schemes, where system performance is degraded severely.

A comparison of four mutual exclusion schemes is listed in Table 1: pure test and set (PTS), Anderson's array-based queuing lock (ADS), Mellor-Crummey and Scott's list-based queuing lock (MCS), and our scheme. Processors in the PTS scheme acquire and spin on the same single lock located at a remote memory module, rendering that memory module to be a severe hot spot. This drawback was partially overcome by the ADS scheme, where each processor spins on a separate remote memory module to relieve contention. However, those spin operations must go through the network, creating considerable traffic. Processors in the MCS scheme spin on local memory module, with the help of linked list redirection, to solve the remote spinning problem. There is still one drawback yet to be addressed: one single remote memory module for acquiring the lock can become a hot spot. Our proposed scheme avoids this drawback by employing a tree of locks to distribute contention and also eliminates unnecessary synchronization traffic due to spinning on the remote memory locations or linked list redirection.

In all previous schemes, up to N processors compete for a single lock, with only one of them getting the lock at a time, while the remaining processors must wait to be served in sequence. In contrast, our proposed scheme employs a tree of locks so that at most n processors compete for a common lock in each group at any time. All locks (i.e., leaf locks, intermediate locks, and the root lock) in our lock tree form / merge CLs in parallel, thereby reducing tremendously the degree of contention at any lock and also forming the waiting queue in parallel. Furthermore, only the header in each group is eligible to compete for the parent lock in the next level, while all members halt and wait for being awakened, resulting in further reduction in contention at the parent locks. In order to gauge the effectiveness of the proposed scheme in reducing hot spot contention, we

TABLE 1
A Comparison of Four Mutual Exclusion Schemes

	PTS	ADS	MCS	Ours
Acquire lock at	single lock	single lock	single lock	tree of locks
Hot spot existing at	single lock	single lock	single lock	no
Spin on	single lock	remote memory	local memory	local memory
Linked list redirection	no	no	yes	no

derive a lower bound on the elapsed time as follows. Our derivation does not take the contention time into account.

4 Bound on Elapsed Time

Consider a system with M processors, among which N processors are involved in mutual exclusion. A mutual exclusion session includes the first processor acquiring the lock, the lock privilege being consigned in sequence to the other $N-1$ processors, and finally the last processor resetting the lock.

The *elapsed time* is defined as the duration of a mutual exclusion session, measured by the time from the first processor issuing an operation to access the lock till the lock being released by the last processor. To obtain a lower bound on the elapsed time (E_ϕ), we consider only the time needed to carry out mutual exclusion activities, ignoring the effect due to contention either in the network or at memory modules. This bound obviously reveals the best scenario a mutual exclusion session can be, as follows:

$$E_\phi = T_1 + T_2 \times (N-1) + T_3,$$

where T_1 is the time for the first processor to acquire the lock, T_2 is the time for a processor to consign the privilege to its next processor, and T_3 is the time to reset the lock.

If T_1, T_2, and T_3 are assumed to take the same amount of time, i.e., the message turnaround time (T_{ta}) between a processor and a remote memory module, the expression for E_ϕ is simplified to $E_\phi = T_{ta} \times (N+1)$. The message turnaround time for a MIN-based multiprocessor can be calculated by

$$T_{ta} = 2 \times \left(T_{sw} \times \lceil \log_m M \rceil\right) + T_{msg} + T_m,$$

where T_{sw}, M, m, T_{msg}, and T_m are the switch latency, the system size, the switch size, the time for a processor to send one message, and the remote memory module latency, respectively. Since T_{sw}, T_{msg}, and T_m are constant, we can further simplify the expression of E_ϕ to

$$E_\phi = \left(C_1 \times \lceil \log_m M \rceil + C_2\right) \times (N+1),$$

where C_1, C_2 are constant values. This indicates that the lower bound on the elapsed time grows by the rate of $\Omega(N \log_m M)$, where N is the number of processors involved in mutual exclusion.

Since the derived lower bound on the elapsed time does not take contention into account, a scheme with performance closes to the derived lower bound indicates that it is more effective in reducing hot spot contention. The performance results in the next section demonstrate that our scheme achieves the best performance among known schemes (in terms of the elapsed time and the response time), as its elapsed time is the closest to the lower bound.

5 Performance Study

For comparison, we simulated our scheme and earlier techniques using an event-driven simulator, called PARSIM, which was developed as a part of the CHIEF simulation environment at CSRD in University of Illinois [4], [5].

5.1 Simulation Model

PARSIM simulates a shared-memory architecture with M processors and M memory modules interconnected by an Omega network. It can handle the system size up to 256. No cache is considered in this simulation study. Any unsuccessful, blocked request in the network is held in its buffer instead of being dropped. Every request consists of three words of 32 bits each, constituting a packet. The network is packet-switched with cut-through pipelining for efficiency.

It is assumed that a remote memory module, on receiving a wake-up request from a processor, sends two acknowledge requests back: one to the processor which issued the wake-up request, and the other to the processor to be awakened. Actually, this assumption is not necessary in such a system as the Butterfly multiprocessor, where processors and memory modules are at the same side of the interconnection network. Each processor in such a system has its local memory.

Four mutual exclusion schemes were simulated: PTS, ADS, MCS, and our scheme. The simulation results were obtained under the same set of system parameters, including

1) the latencies for processors, switches, and memory modules equal to one, two, and ten clock cycles, respectively, and
2) the buffer in a network switch or in a memory module able to accommodate one request (three words).

The switch size is two, so the request turnaround time (T_{ta}) for $M = 256$ is 45 cycles, according to the formula in Section 4.

5.2 Implementation

The ADS scheme requires both fetch_and_store and fetch_and_increment atomic instructions, while the other three schemes employ only the fetch_and_store instruction. In the PTS scheme simulated, each processor repeatedly sends a fetch_and_store request to a lock located in a remote memory module to see if the lock acquisition is successful. Only one processor can hold the lock at a time, and all the other waiting processors keep spinning on the lock until the lock is acquired. On completing the critical section, the processor issues a fetch_and_store instruction to reset the lock. In the ADS scheme and the PTS scheme, the wait-spinning processors issue new spin requests only after previous spin requests have returned, i.e., a waiting processor has at most one outstanding spin request at a time.

Once a processor knows itself being a member processor of the global CL (or list) in our (or the MCS) scheme, it halts, waiting to be awakened. To consign the lock privilege, the processor with the privilege sends a wake-up request to awaken the next processor in the CL (or list). For the ADS scheme, the processor with the lock privilege resets the next processor's spin flag located in a remote memory module to consign the privilege, while the next processor must poll its spin flag to learn privilege consignation. This is because a processor in the ADS scheme knows only its next processor's spin location (rather than its identification number).

In reality, processors tend to arrive at the critical section at different times. In our simulation, the mutual exclusion

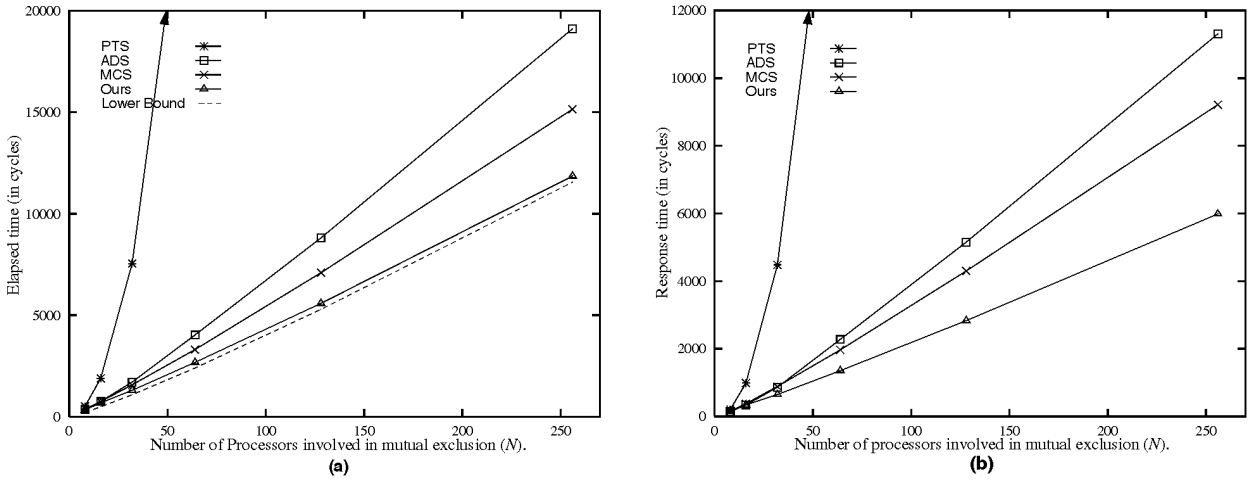

Fig. 10. Performance of four schemes in comparison with the lower bound, which is shown by the dashed curve (with nil critical section time).

requests of processors are characterized by a normal distribution[1] with mean value μ and standard deviation σ, as used in other studies [7], [8]. For a small σ under the normal distribution, all involved processors in the system generate mutual exclusion requests in a short period of time, likely to render the memory module containing the lock a hot spot. The request patterns generated by processors are affected by several factors, such as random delays and nondeterministic processing requirements [12], memory contention [13], the program's internal structure and control dependence graph [6], and so on.

The performance measures of interest are

1) the elapsed time and
2) the response time, which is defined as the average time from acquiring the lock till finishing with the critical section by a typical processor.

A small elapsed time indicates the effectiveness of a scheme in reducing hot spot contention, while a short response time reflects fast lock acquisition. A serious hot spot contention usually leads to many processors waiting in the queue (of the ADS scheme), in the linked list (of the MCS scheme), or in the global CL (of ours), thus prolonging the response time.

5.3 Simulation Results

In our simulation, all processors are involved in mutual exclusion, i.e., $N = M$. The simulation results are obtained by averaging ten runs to arrive at unbiased outcomes. In each run, a different random number seed is given to generate various patterns of the processor request time. For a system with N involved processors, each point of our simulation results shown in the figures is the outcome of collecting N critical section entries, i.e., each processor executes the critical section exactly once instead of repeatedly acquiring the lock. This reflects such real program execution scenarios as Integer Sort (IS) in the NAS benchmark suite [16], where a processor in IS requests the critical section exactly once before facing a barrier.

1. Our normal distribution is generated using a procedure provided by Dr. Seth Abraham at Purdue University.

5.3.1 Overall Performance

In Fig. 10, the elapsed time and the response time versus N are illustrated for the four schemes, where our scheme uses $n = 8$. The lower bound on elapsed time derived using expressions in Section 4 is also included in Fig. 10a for comparison. The mean value μ and the standard deviation σ of request generation patterns are set to 4,000 cycles and 50 cycles, respectively, and the critical section time is assumed nil. The use of nil critical section time in this illustration is to compare these schemes with the lower bound, which is derived by assuming a nil critical section time (although the critical section in reality is not nil).

The PTS scheme, as expected, has the worst performance among all the four. As N increases, both its elapsed time and the response time increase quickly, much faster than those of the other schemes. When compared with the ADS scheme, the MCS scheme exhibits better performance, as shown in the figure. This is because the ADS scheme generates more synchronization traffic over the network (due to spinning) than the MCS scheme (due to linked lock redirection). Our scheme achieves the best performance throughout the range of N simulated, as might be expected, because it avoids any single lock from becoming a hot spot and removes undesirable synchronization traffic caused by spinning (in the ADS scheme) or linked list redirection (in the MCS scheme). In addition, the performance gaps between our scheme and the other schemes tend to increase gradually as N grows. Although the simulator used, PARSIM [4], can handle at most 256 processors, we expect the gaps to keep increasing consistently when N goes beyond 256 due to the fact that hot-spot contention gets more serious as N becomes large.

The effectiveness of our scheme is evident when compared with the lower bound shown by the dashed curve in Fig. 10a. It is observed that the performance curve of our scheme stays closely to the lower bound for every N, with the gap between the two curves kept virtually unchanged. This gap is mainly due to the time needed for processors to accomplish the global lock acquisition through the tree of locks. If a tree of locks with fewer levels (denoted by $\lceil \log_n N \rceil$) is chosen, the global lock acquisition time gets reduced and the gap thus shrinks, but the network and memory contention would increase.

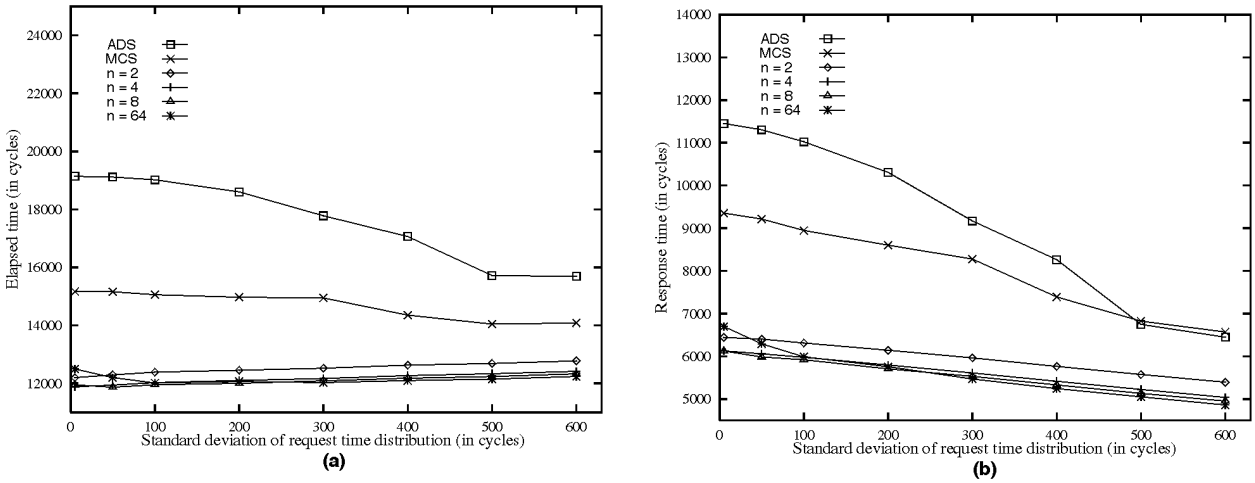

Fig. 11. The impact of standard deviation (σ) on the behaviors of the three schemes (with nil critical section time).

5.3.2 Effect of Standard Deviation (σ) under Empty Critical Section

For a given N, the optimal n depends mainly on σ (standard deviation of processor request time distributions). The performance as a function of σ for $N = 256$ under different schemes is shown in Fig. 11, where the critical section time is assumed nil. The σ values range from five cycles to 600 cycles (which equal 13.3 times of the message turnaround time T_{ta} under $N = 256$). Under the circumstance of $\sigma = 600$ for $N \leq 256$, the distribution of processor request time is quite sparse, as the time interval between the first critical section request and the last one for $\sigma = 600$ and $N = 256$ is about 80 times of T_{ta}. For a given σ, the optimal n in our scheme is decided by the trade-off between the network/memory contention time and the global lock acquisition time. For different σ values, the optimal n varies. From Fig. 11, $n = 8$ achieves the best performance both on the elapsed time and on the response time under $\sigma < 200$ cycles, while the optimal n is 64 for $\sigma \geq 200$ cycles. It appears that, for a given N, the optimal n increases as σ grows. This is somewhat expected, because a larger σ means that processors tend to request mutual exclusion over a longer period of time, leading to a lighter contention in network/memory, and the choice of a larger n thus reduces the global lock acquisition time without causing severe contention.

The impacts of σ on the performance of ADS and MCS schemes are significant, as can be observed from Fig. 11; a smaller σ tends to yield a yield a larger elapsed time and a notably longer response time, as a result of more serious hot-spot contention and longer lock acquisition. Our scheme, however, leads to only a slight decrease in the elapsed time and a slight increase in the response time as σ decreases, indicating that it successfully alleviates hot-spot contention and achieves fast lock acquisition. Consequently, our scheme exhibits superior performance. In general, the number of member processors formed in the queue (of the ADS scheme), in the linked list (of the MCS scheme), or in the global CL (of ours) get reduced as σ grows, resulting in shorter response times of all the three schemes as shown in Fig. 11b. It should be noted that, under $\sigma \geq 500$, processors generate requests quite sparsely and contention is no longer serious. This effect can be observed from Fig. 11b that the ADS scheme has a smaller response time than that of the MCS scheme for such a σ, because processors in the ADS scheme then no longer spend much time spinning to acquire the lock (due to light contention in network/memory), while link list redirection is still needed (in the MCS scheme). The degree of hot spot contention is generally decided by two parameters: the number of involved processors (N) and the standard deviation of processor request time distributions (σ). For a fixed N, the contention gets serious as σ decreases. For a fixed σ, the contention gets severer as N increases.

5.3.3 Effect of Standard Deviation (σ) under Non-empty Critical Section

The critical section in an application program usually contains a few instructions. To investigate the behavior of our scheme applied to real programs, we had carried out simulation under various nonzero critical section times. The performance versus σ for $N = 256$ under the critical section time equal to 10 cycles is shown in Fig. 12. The results for other critical section times follow similar trends. Each curve in the figure has a higher magnitude when compared with the corresponding curve in Fig. 11, and the gaps between the ADS curve and the MCS curve (or our scheme's results) get larger. This is mainly because a non-zero critical section time delays the process of privilege consignation, thus rendering the subsequent processors in the queue of the ADS scheme to generate more synchronization traffic over the network (due to spinning), while each member processor in the MCS scheme (or our scheme) does not generate synchronization traffic over the network, because processors in the MCS scheme (or our scheme) halt and wait for being awakened once they know themselves to be member processors.

6 Concluding Remarks and Future Work

In this paper, an efficient implementation of mutual exclusion locks in multiprocessors has been introduced, which not only avoids serious hot-spot contention inherent to both the MCS and ADS schemes, but also eliminates unnecessary synchronization traffic due to spinning or linked list redirection. Consequently, the proposed scheme achieves

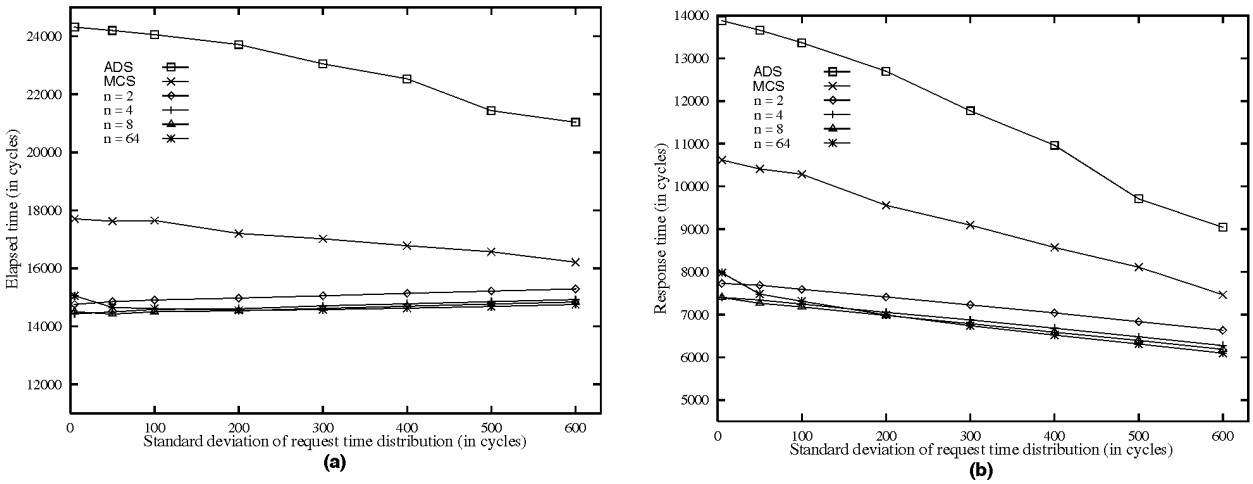

Fig. 12. The impact of standard deviation (σ) on the behaviors of the three schemes (with critical section time equal to 10 cycles).

the best performance among known techniques and yields elapsed time results close to the lower bound. Since the elapsed time and the response time of our scheme grows slowly as the number of processors involved in mutual exclusion increases, it offers good scalability and is suitable for large systems.

Both the ADS and MCS schemes require a single remote lock to construct a queue. For the MIN-based multiprocessor, synchronization traffic is more due to spinning in the ADS scheme than due to linked list redirection in the MCS scheme, suggesting that the MCS scheme exhibits a better performance than the ADS scheme, as confirmed by our experimental results shown in Fig. 10. For a bus-based multiprocessor with cache, however, each waiting processor in the ADS scheme can spin on a counter located in a local cache line (without producing synchronization traffic over the bus), while the MCS scheme still requires linked list redirection done over the bus, indicating that the ADS scheme is superior. This has been validated by experimental results obtained on a Sequent Symmetry Model B machine [11]. When our scheme is applied to bus-based multiprocessors, it would still exhibit the best performance due to dispersing a single remote lock into a tree of remote locks, eliminating hot-spot contention.

In our proposed scheme, there is a possibility for the global header to repeat the process of closing a global linked list into a global CL (i.e., circular list) several times before releasing the lock, when a global linked list coexists with the global CL. Under such a situation, the global header acts very much like a centralized controller to consign the privilege to the next processor in the global CL without entering the critical section itself. However, the global header changes from time to time and it, in fact, can take advantage of the lock privilege arrival by entering the critical section immediately (if it is requesting for the critical section) on receiving the lock privilege, and then consigning the privilege to the next processor after finishing with the critical section. With this, the global header enters the critical section even without sending any request message out.

This paper considers the situation that every processor requests mutual exclusion once before encountering barrier synchronization. This reflects such real program execution scenarios as Integer Sort (IS) in the NAS benchmark suite [16]. Future research on this topic includes developing synchronization primitives based on the proposed mutual exclusion scheme and evaluating the performance of representative application programs on top of the developed primitives. In addition, it is interesting to investigate situations where every participating processor generates many mutual exclusion requests before encountering any barrier synchronization.

ACKNOWLEDGMENTS

A preliminary version of this paper was presented at the Ninth International Parallel Processing Symposium, April 1995. This material is based upon work supported in part by the U.S. National Science Foundation under Grants MIP-9201308 and CCR-9300075 and by the State of Louisiana under Contract LEQSF(1993-95)-RD-A-35.

REFERENCES

[1] N. Koike et al., "NEC Cenju-3: A Microprocessor-Based Parallel Computer," *Proc. Eighth Int'l Parallel Processing Symp.*, pp. 396-401, Apr. 1994.

[2] D.J. Kuck et al., "Parallel Supercomputing Today and the Cedar Approach," *Science*, vol. 21, pp. 967-974, Feb. 1986.

[3] G.F. Pfister and V.A. Norton, "'Hot-Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Trans. Computers*, vol. 34, no. 10, pp. 943-948, Oct. 1985.

[4] J.D. Bruner, H. Cheong, A. Veidenbaum, and P.C. Yew, "Chief: A Parallel Simulation Environment for Parallel Systems," *Proc. Fifth Int'l Parallel Processing Symp.*, pp. 568-575, Apr. 1991.

[5] C.J. Beckmann, "CARL: An Architecture Simulation Language," CSRD Report No. 1066, CSRD, Univ. of Illinois, Urbana, Jan. 1991.

[6] V. Sarkar, "Determining Average Program Execution Times and their Variance," *1989 Proc. SIGPLAN Notices Conf. Programming Language Design and Implementation*, vol., 24, no. 7, pp. 298-312, 1989.

[7] S. Abraham and K. Padmanabhan, "Performance of the Direct Binary n-Cube Network for Multiprocessors," *IEEE Trans. Computers*, vol. 38, no. 7, pp. 1,000-1,011, July 1989.

[8] M.C. Wang, H.J. Siegel, M.A. Nichols, and S. Abraham, "Reducing the Effect of Hot Spots by Using a Multipath Network," *Proc. 1993 Int'l Conf. Parallel Processing*, vol. I, pp. 274-281, Aug. 1993.

[9] N.-F. Tzeng, "A Cost-Effective Combining Structure for Large-Scale Shared-Memory Multiprocessors," *IEEE Trans. Computers*, vol. 41, no. 11, pp. 1,420-1,429, Nov. 1992.

[10] T.E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6-16, Jan. 1990.
[11] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Computer Systems*, vol. 9, no. 1, pp. 21-65, Feb. 1991.
[12] V. Adve and M.K. Vernon, "The Influence of Random Delays on Parallel Execution Times," *1993 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 61-73, May 1993.
[13] M. Dubois and F. Briggs, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," *IEEE Trans. Software Eng.*, vol. 8, no. 4, pp. 419-431, July 1982.
[14] J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 64-75, Apr. 1989.
[15] P.-C. Yew, N.-F. Tzeng, and D.H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 388-395, Apr. 1987.
[16] D. Bailey et al., "The NAS Parallel Benchmarks," Technical Report TR RNR-91–002, NASA Ames, Aug. 1991.

Shiwa S. Fu received the BS degree in information and computer engineering from Chung-Yuan University, Taiwan, in 1986, and the MS degree in computer science from the State University of New York at Binghamton in 1990. Prior to joining SUNY-Binghamton, he was a system engineer with WANG Industrial Corp., Taipei, Taiwan, and an assistant engineer with the Industrial Technology Research Institute (ITRI), Taiwan.

Currently, he is a PhD candidate in computer science with the Center for Advanced Computer Studies, the University of Southwestern Louisiana, Lafayette. His research interests include parallel processing, distributed computing, and operating systems. He is a member of the IEEE Computer Society.

Nian-Feng Tzeng (S'85-M'86-SM'92) received the BS degree in computer science from National Chiao Tung University, Taiwan, the MS degree in electrical engineering from National Taiwan University, Taiwan, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1978, 1980, and 1986, respectively.

He has been with the Center for Advanced Computer Studies at the University of Southwestern Louisiana (USL), Lafayette, since June 1987. From 1986 to 1987, he was a member of technical staff, AT&T Bell Laboratories, Columbus, Ohio. He is on the editorial board of *IEEE Transactions on Computers*, has served on program committees of several conferences, and is a distinguished visitor of the IEEE Computer Society. He is a co-guest editor of a special issue of the *Journal of Parallel and Distributed Computing* on distributed shared memory systems, 1995. His research interests include parallel and distributed processing, high-performance computer systems, high-speed networking, and fault-tolerant computing.

Dr. Tzeng is a member of Tau Beta Pi, a member of the ACM, and the recipient of the outstanding paper award of the Tenth International Conference on Distributed Computing Systems, May 1990. He received the University of Southwestern Louisiana Foundation Distinguished Professor Award in 1997.