Concurrent Updates of Events List for Parallel VHDL Simulations

M. Arshad
AT&T Bell Laboratories
Columbus, Ohio 43213

J. E. DeGroat
The Ohio State University
Columbus, Ohio 43210

Abstract

VHDL simulations running on shared memory multiprocessors have a global events list. The list is updated by multiple processors. In a traditional implementation, a processor locks the entire list structure, updates the list, and then unlocks the structure. We present data structures and algorithms that allow multiple processors to update the list concurrently. On processors with compare&exchange read-modify-write capability the problem is trivially simple and does not require a lock at all. On processors that do not have compare&exchange capability, events are inserted concurrently except when two or more processors attempt to insert an event between the same two elements of the list. A local lock is used to resolve this contention and let one processor insert the event at a time. Another processor inserting an event between any other two elements in the list can still do so concurrently with the first processor. Using exchange read-modify-write capability, we provide contention free busy-waiting for the processors that may have to wait for the same local lock. We further ensure that the waiting processors are served in a FIFO manner. The entire linked list structure is not locked under any circumstances. This will greatly help performance first by allowing concurrent updating and second by using a contention free local lock when a lock is needed.

1 Introduction

As shown in Figure 1, the events list in VHDL simulations is a two dimensional singly linked list. All concurrent events occurring at the same time are linked in a singly linked list of concurrent events and these lists of concurrent events themselves are linked in a singly linked list according to the time at which the events are to be processed. A set of concurrent events that occurs prior to another set is linked ahead of the other set. The order among the concurrent events occurring at the same time is unimportant while the order among the lists of concurrent events occurring at different times is important and must be maintained.

In a traditional implementation when a processor attempts to post a new event, as a first step, it may lock the entire linked list structure. Then it may search through the time based ordered linked list to find where

which it is to be processed. Once the event is inserted in the linked list, the processor inserting the event unlocks the linked list structure. The new event starts a new list of concurrent events. Any new events occurring at the same time as this event, will be inserted in this new list of concurrent events.

![Time Based Ordered Linked List]

**Figure 1. Linked Lists of Events in VHDL Simulations**

When all processing for a given time has completed, the time is advanced to the next time in the time based ordered linked list. This is done by one of the processors which also takes the list of concurrent events at the next time and dispatches events from that list starting another cycle of execution [8].

To lock the entire linked list structure is overly conservative and degrades performance since only one processor is granted access mutually excluding the rest of the processors. The act of mutual exclusion is to make sequential the otherwise concurrent processing of the parallel processors. The mutual exclusion should be minimized if not eliminated, to take full advantage of the processing power of the parallel processors. We present data structures and algorithms that will do just that.

2 Concurrent Updates of Events List

Given a linked list, two processors may concurrently insert an event if each inserts its event at a different place in the linked list. As an example, if a processor inserts an event between elements marked T₀ and T₁ in Figure 1, and another processor inserts an event between any other two elements except between T₀ and T₁, they can do so concurrently since they do not update the same element. A global lock, locking the entire linked list structure, therefore, is never needed. Without using any lock a processor can search through the linked list to find where in the linked list it must insert the event. It will encounter one of the following two situations.

1. It needs to insert the event in the time based ordered linked list. This does not require a lock on processors with compare&exchange (also known as compare&swap) read-modify-write capability. However, on processors that do not have the compare&exchange capability, a local lock is required to insert the events in an orderly fashion. As an example, when a processor needs to insert an event between elements marked T₁ and T₂, it must lock the element marked T₁. It implies a local lock is associated with each element in the time based ordered linked list.
2. It needs to insert the event in an already existing list of concurrent events. This does not require a lock on processors that, at a minimum, has exchange (also known as swap) atomic read-write capability. Most all the modern CISC as well as RISC processors provide exchange capability. However, if exchange is not available, a local lock may be used. This local lock is associated with the head element of the list of concurrent events. The head element is also a part of the time based ordered linked list and needs a local lock for inserting an event in the time based ordered list. One may use a single lock or two independent locks. Two locks will allow insertion of an event in the list of concurrent events anchored at an element $E_i$ in parallel with insertion of an event in the time based ordered list after $E_i$.

In what follows we first present a simple solution which is based on compare&exchange capability and does not require a lock. Next, we present a solution for the case in which a local lock is required to insert an event in the time based ordered linked list but does not require a lock for inserting an event in the list of concurrent events. This solution can be extended to cover the case in which a local lock is also required to insert an event in the list of concurrent events; however, we will not discuss it in this paper.

2.1 A Lock Free Solution

On processors with compare&exchange or its equivalent instructions, the events list can be updated without a lock. The data structure is shown in Figure 2 below. Events occurring at the same time are linked in singly linked lists which themselves are linked in a time based ordered singly linked list. Head points to the first element of the time based ordered singly linked list. Each element is a structure that consists of four members. List points to the singly linked list of concurrent events. It is NULL in all the elements in the list of concurrent events except the first one which serves as the head element of the concurrent list and is also linked in the time based ordered list. Data is either a record or a pointer to a record containing information about the event itself. Time is the time at which the event is to occur and next points to the next element in the time based ordered list.

![Figure 2. Data Structure for Concurrent Updates Using Compare&Exchange](image)

Using compare&exchange the algorithm is trivially simple. The algorithm is non-blocking [7]. It is presented in Figure 3 in C. It takes a pointer to Head and a pointer to the event being inserted as parameters. It uses
two local variables this and next. Variable this is initialized to point to Head and next gets initialized (first time within the loop) to the contents of Head which may be NULL if the list is empty or point to the first element in the list otherwise. Later, this points to the next member of the structure of an element while the local variable next (after the execution of the first statement in the loop) points to the next element in the list.

Using this and next local variables, we traverse the linked list till we find the location in the time based ordered list at which the event is to be inserted. This is accomplished in the first six lines within the loop; up to and including the continue statement as shown in Figure 3. At the end of this search i->time, the time at which the event being inserted is to occur, is either equal to or less than next->time, the time of the next event in the time based ordered list. If i->time is equal to next->time, the event is to be inserted in the list of concurrent events pointed to by local variable next, i.e., it is linked at next->list and the current event at next->list is linked behind the event being inserted. Next four lines including the break statement, accomplish this; we break out of the loop since the event has been successfully inserted.

If we reach the statement following the break statement, either next is NULL, i.e., we have reached the end of the list, or i->time is less than next->time. In either case the event is to be inserted at the member structure next of the previous element, i.e., the location pointed to by this. We copy the contents of local variable next into the structure member next of the event being inserted and using compare&exchange attempt to update the location pointed to by this. The attempt will succeed if no other processor has linked an event at this in between this processor read its contents on the first line of the loop and executed the compare&exchange. If the attempt succeeds, the event is inserted and the loop terminates.

Compare&exchange returns the current value of memory pointed to by this. The memory is updated with the new value i, a pointer to the event being inserted, only if its contents are equal to next. If the attempt fails as indicated by the return value of compare&exchange not being equal to next, some other processor succeeded in inserting an event at this. The attempt by this processor has failed and it must re-try. It re-reads the contents of this at the first line of the loop makes another attempt.

```c
insert_event(struct event *H, struct event *i)
{
    struct event **this, next;
    this = H;
    do {
        next = *this;
        if (next != NULL) {
            if (i->time > next->time) {
                this = &next->next;
                continue;
            }
            if (i->time == next->time) {
                i->next = exchange(&next->list, i);
                break;
            }
        }
        i->next = next;
    } while(compare&exchange(this,next,i) != next);
}
```

Figure 3. Algorithm Using Compare&Exchange
2.2 A Solution Using Local Locks

Processors that do not have compare&exchange capability can update the list of concurrent events without a lock if they have exchange capability, but require a local lock to update the time based ordered list. When a processor waits for a short duration, it uses a lock to busy wait. A traditional lock in which a processor continually polls the lock variable to determine whether the lock is free can severely degrade the performance [2][3][6][10][11][12][13]. The extent of contention for a lock will determine the impact. By using a local lock the contention, on the average, may reduce by the number of elements in the time based ordered linked list if the new events are equally distributed over the entire time based ordered linked list. This, however, may not be a realistic assumption in some event driven systems. Consequently, the contention may not reduce by as much but it reduces nevertheless.

Locks that are frequently contended by more than one processor may use algorithms that minimize or eliminate contention[1][2][5][6][10][11]. Except for the linked list based lock developed by Mellor-Crummey and Scott [10][11], the rest of the contention free locks require an array equal in size to the number of processors for each lock. Also, they rely on coherent caches to minimize or eliminate contention. Since we use a local lock which is associated with each element, array based locks are not as attractive. Mellor-Crummey and Scott's link based lock uses one long word per lock. It eliminates contention on processors with or without coherent caches as long as processors without coherent caches have some local shared memory'. However, it requires compare&exchange to ensure FIFO behavior. The version that uses exchange does not guarantee the FIFO behavior. We have developed a linked list based lock that uses two long words per lock. It preserves the FIFO behavior for the waiting processors using exchange as the only read-modify-write capability [4]. Otherwise, it has the same characteristics as the lock developed by Mellor-Crummey and Scott. We will use our lock in this application. We provide more details about the lock in the next sub section.

A lock is associated with each element and with Head. The overall data structure remains the same as shown in Figure 2. However, we redefine Head and the structure member next to be a structure that consists of two elements link and lock as shown in Figure 4. The lock itself consists of two elements head and tail. In addition to the lock which is associated with each element and may be in the global memory, it requires a data structure in the local shared memory that consists of two elements. A processor will pass this data structure labelled as proc as an additional parameter to the function implementing the algorithm.

![Figure 4. Data Structure Modification for Local Lock](image)

The algorithm is modified and presented in Figure 5 in C. As a first step, the lock associated with the event being inserted is initialized to free. The next part of the algorithm that traverses the time based ordered list is conceptually not changed; this is replaced by this->link because of the change in the data structure. Likewise, there is no change to the part of the algorithm that inserts the event in the list of the concurrent

---

1. Local shared memory is the memory that is local to one processor but directly accessible from other processors. The processor to which the memory is local can access it locally without generating requests for the global bus while other processors can access it over the bus.
events. The part of the algorithm that inserts the event in the time based ordered linked list is what is changed; we describe this change next.

```c
void
insert_event(struct link *H, struct event *i, struct proc *p)
{
    struct link *this;
    struct event *next;
    struct lock *lock;

    i->next.lock.head = NULL;
    i->next.lock.tail = &i->next.lock.head;
    this = H;
    do {
        next = this->link;
        if (next != NULL) {
            if (i->time > next->time) {
                this = &next->next;
                continue;
            }
            if (i->time == next->time) {
                i->next.link = exchange(&next->list,i);
                break;
            }
        }
    }

    lock = &this->lock;
    acquire(lock,p);
    if (this->link == NULL || this->link->time > i->time) {
        i->next.link = this->link;
        this->link = i;
        release(lock,p);
        break;
    }
    release(lock,p);
} while (TRUE);
```

Figure 5. Algorithm Using a Local Lock

When a processor needs to insert an event in the time based ordered list, it attempts to lock the element after which it is going to insert the event. It will either acquire the lock immediately or will busy wait spinning in its local shared memory or cache till it gets the lock. In either case when it reaches the statement following the `acquire()` function call, it has acquired the lock. Another processor trying to insert an event at the same point in the linked list may have succeeded in inserting the event ahead of this processor. This is possible even when the processor acquires the lock immediately since a processor may have acquired the lock, inserted the event, and unlocked in between this processor read the value this->link on the first line within the loop and acquired the lock. Therefore, the processor must check whether it is at the right point in the time based ordered linked list for inserting its event after having acquired the lock. If it is, it inserts the event, unlocks, and breaks out of the loop. Otherwise, it releases the lock and continues to search for the right place in the time based ordered linked list starting from this element to make another attempt.
2.3 A Contention Free Local Lock

The list based lock that we have developed uses two long words per lock and ensures the FIFO behavior using *exchange* as the only atomic read-write capability. In addition to the lock which is a structure that consists of head and tail and is associated with each element of the list, it requires a structure in the local shared memory of each processor. This structure consists of two elements **succ** and **spin**, and each processor declares it in its local shared memory or on a separate cache line. A processor needs only one such structure since it waits for only one lock at a time. Formally, the data structure are as shown in Figure 6.

```
struct lock {
    struct proc *head;
    struct proc **tail;
};
```

```
struct proc {
    struct proc *succ;
    int spin;
};
```

**Figure 6. Data Structures for our Contention Free Lock**

The algorithm to acquire the lock is shown in Figure 7. To acquire the lock, a processor sets its **succ** element to point to itself to indicate that there is no processor linked behind it and sets **spin** to *WAIT* to prepare itself for busy waiting. Next, it exchanges the address of its **succ** element with the contents of tail; this is done by the inner *exchange*() which is the first parameter to the outer exchange in the if statement. As a result, tail points to its **succ** element and the value returned by the inner *exchange*() either points to head or the **succ** of its predecessor, i.e., the last processor requesting the lock prior to this processor. In either case it provides the address where this processor can link itself. If tail pointed to head and head was **NULL** before this processor requested the lock, the lock is free and this processor does not need to wait. In all other cases this processor must wait.

```
void acquire(struct lock *L, struct proc *p) {
    p->succ = p; p->spin = WAIT;
    if (exchange(exchange(&L->Tail, &p->succ), p) != NULL) 
        while(p->spin == WAIT) { }; /* busy wait */
}
```

**Figure 7. Algorithm to Acquire the Lock**

If tail was pointing to head, and head was not **NULL**, the value returned by the expression is not **NULL** and the processor waits. If tail was pointing to another processor, the outer *exchange*() in the if statement links this processor behind the other processor and returns what was there in the **succ** element of the other processor, i.e., its predecessor. Since this processor must wait, the value returned from the **succ** element of the predecessor must not be **NULL**. This is the reason each processor initializes its **succ** element to a value different than **NULL**. When the lock is not free, the processor waits spinning on its local variable till its predecessor releases the lock and changes the **spin** variable of this processor to **FREE**.
The algorithm to release the lock is presented in Figure 9. A possible sequence while releasing the lock is graphically shown in Figure 8. To release the lock, a processor first checks whether it has a successor. When L->tail points to the releasing processor, there are no successors waiting for the lock. Further, we can be sure that the head element of the lock is not in use. The releasing processor points head to itself, exchanges the address of head with the contents of tail, and links head at the address returned by the exchange; this all happens on the single line in the first if statement. The address returned by the exchange() is the address of the succ element of the last processor, i.e., P_i in Figure 8, that requested the lock before tail is switched to point to head. This may be the processor releasing the lock itself or another processor that may have requested the lock in between this processor checked for a successor and executed the exchange().

Next, this processor waits for its succ element to be updated. If no other processor requested the lock, the succ element of this processor points to head and the while() loop immediately terminates. If a successor has requested the lock either before the check, i.e., before the if statement, or before the exchange() statement, the while() loop ensures that the successor has properly linked behind this processor.

```c
void release(struct lock *L, struct proc *p)
{
    if (L->tail == &p->suc)
        *(exchange(&L->tail, (L->head=&L->head))) = &L->head;

    while (p->suc == p) {} /* wait for suc to be updated */
    if ((p->suc != &L->head) ||
        ((p->suc = exchange(&L->head, NULL)) != &L->head))
        p->suc->spin = FREE;
}
```

After ensuring the succ is properly updated, the processor checks if head is linked behind it. If it is not, as indicated by the first expression in the second if statement, a successor is waiting for the lock. This processor passes the lock on to the successor by executing the if statement. The second expression in the if statement does not execute; it executes only if the first condition is not true, i.e., the succ element of this
processor points to head. In that case, the processor releasing the lock attempts to free the lock by exchanging NULL with the contents of head. If the head was still pointing to itself, it is changed to NULL, the lock is free, and no further action is needed; the if statement does not execute. However, if the exchange() returns a value other than the address of head, it points to a successor waiting for the lock. The processor releasing the lock passes the lock on to the waiting successor by executing the if statement.

3 Comments and Conclusions

In this paper we have presented techniques to update events list as concurrently as possible given the underlying hardware capabilities. The technique is applicable not only to parallel VHDL simulations but also to any event driven system in general. The use of a local lock allows multiple processors to concurrently insert events. It is a simple concept that allows concurrent insertion of as many events as there are elements in the time based ordered list. Only when more than one processors attempt to insert an event at the same point in the time based ordered list, the lock is contended.

We use a contention free local lock to allow one processor to insert the event at a time. The overhead to acquire this or any other lock with similar characteristics is more than the overhead to acquire a simple bit test-and-set lock. However, the waiting processors spin in their local shared memories and do not generate the repeated read or read-modify-write requests over the global bus. If the lock is seldom contended, the overhead may offset the benefit of using such a lock and a bit test-and-set lock with exponential back off may provide better performance [1] [2]. However, if a lock is even very moderately contended the list based lock presented here will greatly outperform the simple lock.

In our list based contention free lock as well as in the list based lock of Mellor-Crummey and Scott[10][11], the data structure on which the processor waits is either in processor's local shared memory or uses a separate cache line in a cache coherent system. This enables a waiting processor to busy wait locally without generating any traffic over the global bus. Local shared memory is generally available in bus based shared memory multiprocessors. However, a system that has neither local shared memory nor coherent cache can not use contention free locks as described here. It may use a bit test-and-set lock with exponential back off as proposed by Anderson [1] [2].

We have used a linear linked list and a linear search in our discussion throughout this paper. The technique can be easily adapted to other data structures and other search techniques. There is nothing in the algorithm that is dependent on the linear linked list data structure or the linear search technique.

In a system with preemptive scheduling, the interrupts should be disabled before acquiring the lock and enabled after releasing the lock. The functions acquiring and releasing the lock are a preferable place to enable and disable the interrupts respectively. This is important since we do not want a processor to acquire the lock and then be preempted.

To keep the code simple and uniform, we have used two exchange operations while acquiring the lock. The first exchange(), i.e., the inner one, is always required. The second exchange(), however, is required only when tail points to head and head is not NULL. This can be recognized by checking the return value of the first exchange() and reading the value of head. In all other cases the second exchange() can be replaced by a simple write. Likewise, there are two exchange operations in the code to release the lock. Neither exchange() executes if a successor is waiting when a processor checks for a successor. The first exchange() executes if, at the time of checking, no processor is waiting. The second exchange() executes when head is linked behind a processor. Again, this is required only when head is linked behind a processor and head points to itself. If head already points to another valid pointer instead of pointing to itself, a successor is waiting, and the following exchange() is not needed. This can be checked by first reading head. Systems in which exchange is considerably more expensive than simple reads and writes can make these changes to optimize the performance at the expense of some extra code. Also, if the code is actually written in a higher level language, e.g., C, type casting of pointers in some instances is needed.
References


