Multiple Reservations and the Oklahoma Update

Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek
IBM T.J. Watson Research Center

A multiple reservation approach allows atomic updates of multiple shared variables, and simplifies concurrent and nonblocking codes for managing shared data structures such as queues and linked lists. This method can be implemented as an extension to any cache protocol that grants write access to at most one processor at a time.

Multiprocessor computers provide instruction-set support for updating shared variables consistently.[1] Without such support, processors that modify shared variables concurrently risk failure if the updates leave the variables in an inconsistent state. In the most common support technique, each process obtains exclusive access to a set of shared variables, reads those variables and updates them while holding exclusive access, and then releases the exclusive access. The region of exclusive access is called a *critical section*, and the process typically obtains access through control variables called *semaphore* or *lock* variables. Special machine instructions, such as Test-and-Set, control the modification of semaphores to assure that they are updated consistently, and thereby guarantee that at most one processor can execute instructions from a critical section at one time. The special semaphore-based instructions update a single variable *atomically* by performing an uninterruptible Read/Modify/Write sequence of operations. By definition, no other processor can alter the operand between the Read and Write of an atomic update.

However, the use of critical sections can lead to performance degradation. A single processor that is busy updating shared variables can keep many processors waiting at the entrance to a critical section. Moreover, if a processor in a critical section should be suspended by a page fault, context swap, processor crash, or similar mechanism, then no other waiting processor can do useful work in the interim. Consequently, the multiprocessing community is very interested in updating schemes that are free of critical sections.[2]

1063-6552/93/1100-0058 $3.00 © 1993 IEEE

A technique based on *reservations* is perhaps the most efficient general method for updating a *single* shared variable. (The terminology and implementations differ among the machines that use this approach, so we describe the scheme here in a generic form whose details are not necessarily identical to any particular implementation. Our description can easily be adapted to specific machines.) Two instructions use a special reservation register in each processor to perform atomic updates of a single variable:

- *Read-and-Reserve* copies a variable from memory into a general register, and reserves the address by copying it into a special reservation register and marking that reservation as valid.
- *Write-if-Reserved* tests the reservation register, and if the reservation is still valid, updates the shared variable in memory. If the reservation is not valid, the instruction aborts the update.

The reservation register in a processor, say processor A is invalidated when any other processor changes the value of the reserved variable, or when processor A executes a Write-if-Reserved. So, Write-if-Reserved updates the variable only if its original value (read by Read-and-Reserve) is the most recent value of that variable in the multiprocessor system. A valid reservation thus indicates that the updated value was computed correctly because it was based on the shared variable's current value.

Write-if-Reserved returns a condition code to indicate whether or not it performed the update. A program can test the condition code and repeat the attempted update if the code shows that the most recent attempt failed.

Many processors can execute concurrently and consistently when trying to update a single variable protected by these two instructions. Consequently, system performance is less sensitive to context swaps and page faults for such codes than for codes based on critical sections. Later we show that performance may also be better because the effective length of a critical section can be shorter with a reservation-based scheme than with semaphore-based operations.

An update changes all shared variables or none of them, and any change is made atomically so that no other processor can modify the shared variables once the update has begun and until the update has ended.

In this article, we extend the notion of reservations to support the atomic modification of *multiple* shared variables. An update changes all shared variables or none of them, and any change is made atomically so that no other processor can modify the shared variables once the update has begun and until the update has ended. We call this an *Oklahoma update*, a reference to the song "All er Nothin'" from the Rodgers and Hammerstein musical *Oklahoma!*.

This concept has also been developed concurrently and independently by Herlihy and Moss in their work on transactional memory,[3] although many implementation details differ. Our implementation is deadlock-free, assures progress in the absence of false sharing and context swaps, and supports livelock-avoidance (using an exponential back-off mechanism). There is also an option to restart the atomic update immediately when a change to a shared variable guarantees that the Write-if-Reserved will fail.

Atomic updates and fault tolerance are also assured by the *stable storage* structure of Lampson[4] and others,[5] but this typically requires a separate external module. Instead, we show how to implement atomic updates within conventional microprocessors by using existing cache-coherence protocols together with multiple reservation registers and some associated functions. While using a separate module might provide additional fault tolerance, the additional cost of our implementation is negligible for a multiprocessor with hardware support for cache coherence.

Architectural support for concurrent programming without critical sections goes back to the Compare-and-Swap instruction, which provides a way to break an atomic Read/Modify/Write sequence across several instructions. Compare-and-Swap has been implemented on the IBM 370 architecture, among others. A program reads a shared variable to a local register, computes a new value of that variable in a second local register, and then executes Compare-and-Swap, which compares the current value of the variable in memory to the register copy of the original value. If the values are equal, the instruction completes by writing the updated local copy back to main memory. Compare-and-Swap supposedly assures the

atomicity of the Read/Modify/Write sequence because the shared variable's modified value is written to memory only if the original value equals the current value, thus indicating that another processor has not modified the variable in the interim.

However, it is not sufficient to base the update only on the variable's original and current values. If the value is originally A, and other processors meanwhile change it to B and then back to A, then Compare-and-Swap performs the memory update, even though it could lead to inconsistent results in many programs.[6] This problem — often called *the ABA problem* — can arise when one processor examines a pointer, a second processor deallocates the storage accessed by that pointer, and a third processor reallocates new storage through that pointer. The allocation strategy happens to reuse the most recently discarded storage so that the pointer to storage returns to its original value, even though the storage is newly allocated. The storage state seen by the first processor through the pointer is a combination of two different valid storage states. The Compare-and-Swap could succeed when, in fact, it should fail.

If Compare-and-Swap were defined to succeed only if the shared variable's original value is held continuously until the point of update is reached, the ABA problem would not arise. The reservation mechanism provides such a solution to the ABA problem. The implementation of a single reservation is due to Jensen, Hagensen, and Broughton,[7] who proposed instructions essentially the same as Read-and-Reserve and Write-if-Reserved. Their basic ideas are implemented in the load-linked and store-conditional instructions on the MIPS R-4000,[8] in the load-locked and store-conditional instructions on the DEC Alpha,[9] and in the load-and-reserve and store-conditional instructions on the IBM PowerPC.[10]

In the rest of this article, we discuss the implementation of the Oklahoma update and the extensions of automatic restart and livelock avoidance. We also offer programming examples illustrating the application of multiple reservations in typical situations.

The major reason to provide multiple reservations is to remove critical sections from concurrent programs when possible. A critical section is a serial bottleneck. It limits the amount of useful concurrency available in a multiprocessor.

Single-reservation schemes

Current implementations of reservation-based updates have a single reservation associated with each processor, and support the two special instructions Read-and-Reserve and Write-if-Reserved. Different processors can have reservations for the same shared variable. A processor loses its reservation if, before it executes a Write-if-Reserved, another processor modifies the variable.

The correctness of updates that use these instructions relies on the cache-coherence protocol, which must serialize the writes to each location and guarantee that no processor can observe the writes to a particular location out of order. The privilege to write to a location passes from one processor to another sequentially, and thus the writes to that location are properly serialized. When a processor has write privilege for a location, we say it is the *owner* of that location. In multiprocessor cache implementations, ownership is recorded in state bits associated with each cache line.[11] Any cache protocol that grants write access for a shared variable to at most one processor at a time, such as the Berkeley and Firefly protocols,[11] can be extended by our method to support multiple reservations. We assume here that there is an underlying cache-coherence protocol with a method for controlling and passing write privilege. There are many acceptable protocols, so we do not limit our discussion to any particular one.

A processor need not request write privilege when the Read-and-Reserve is executed. (Implementations can differ on this characteristic.) However, when the Write-if-Reserved is executed, if the processor does not already own the shared variable, it must obtain write privilege before the write can be completed.

When processor A requests write privilege, there are two possible outcomes:

- The current owner, processor B, grants A write privilege, and A continues normal execution of Write-if-Reserved.
- Processor B sends a sequence of two messages: one stating that the shared variable's reserved value has

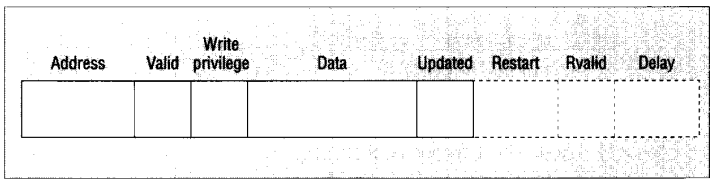

Figure 1. Structure of a reservation register.

been modified, and a second granting write privilege. This occurs when B tries to update the same shared variable concurrently, and has write privilege. Because B can perform the update, it does so, and the cache-coherence protocol communicates the change to other processors. B subsequently receives A's request for write privilege, which it grants. B's two messages to A must remain in order; if they were received out of order, A could make an erroneous update.

This mechanism assures correctness because a processor that owns a variable necessarily holds that variable's most recent value. If a processor does not own a reserved variable before executing Write-if-Reserved, then by the time the processor receives write privilege while executing Write-if-Reserved, either the reservation will have been invalidated or the processor will have confirmation that the reserved value is the most recent. An update based on the most recent value of a shared variable is deemed correct.

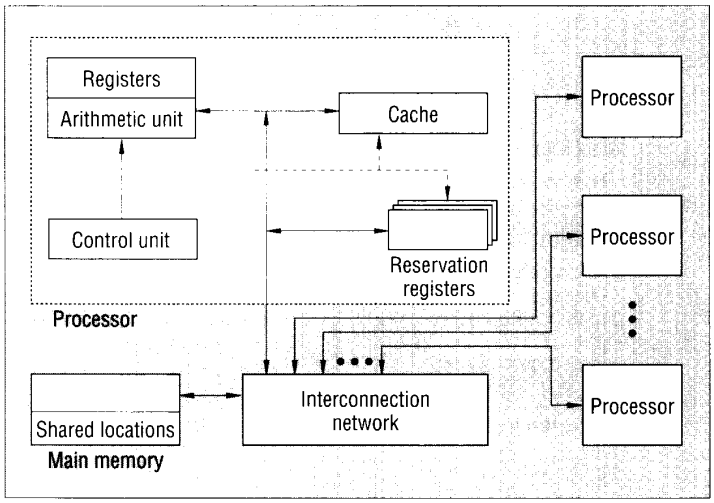

Figure 2. Multiprocessor organization with multiple reservation registers per processor.

The multiple-reservation scheme

The extension of these notions to two or more reservations uses three special instructions instead of just two, and a reservation register with five fields and three optional fields, as shown in Figure 1. (We later discuss the optional fields *restart*, *rvalid*, and *delay*.) The structure of a processor with multiple reservation registers appears in Figure 2; the reservation registers are included with the cache structure to indicate that they are extensions to cache memory.

The *Read-and-Reserve* instruction loads a shared variable into a specified general register, and places a reservation on the variable in a specified reservation register. Executing this instruction initializes the reservation register by making the reservation valid, saves the address of the shared variable in the register, clears the data field that holds an updated value, and clears the update bit to indicate that the data field has not received a modified value. The register also has a write-privilege bit to indicate if the processor has write privilege for the variable; the bit is set if the processor receives write privilege while executing this instruction, or had write privilege before executing it. The update bit and the data field of the reservation register do not exist in current implementations of reservation registers.

The *Store-Contingent* instruction tentatively updates one shared datum. It copies the contents of a specified general register into a specified reservation register data field, and changes the update bit to show that the data field has been updated. The Store-Contingent instruction does not require ownership to execute. When it is executed, the cache-coherence protocol does not communicate a change in a variable to other processors. All data to be updated atomically by an Oklahoma update are updated first in their reservation registers by means of this instruction.

The *Write-if-Reserved* instruction specifies a set of reservation registers and performs an Oklahoma update of the variables reserved by those registers. To do so, it obtains write privilege for all specified reserved variables. If write privilege is obtained for all of them and the reservations remain valid, then the instruction updates in storage the variables for which data in the specified reservation registers were modified. Once the update has begun it is done atomically so that no other processor

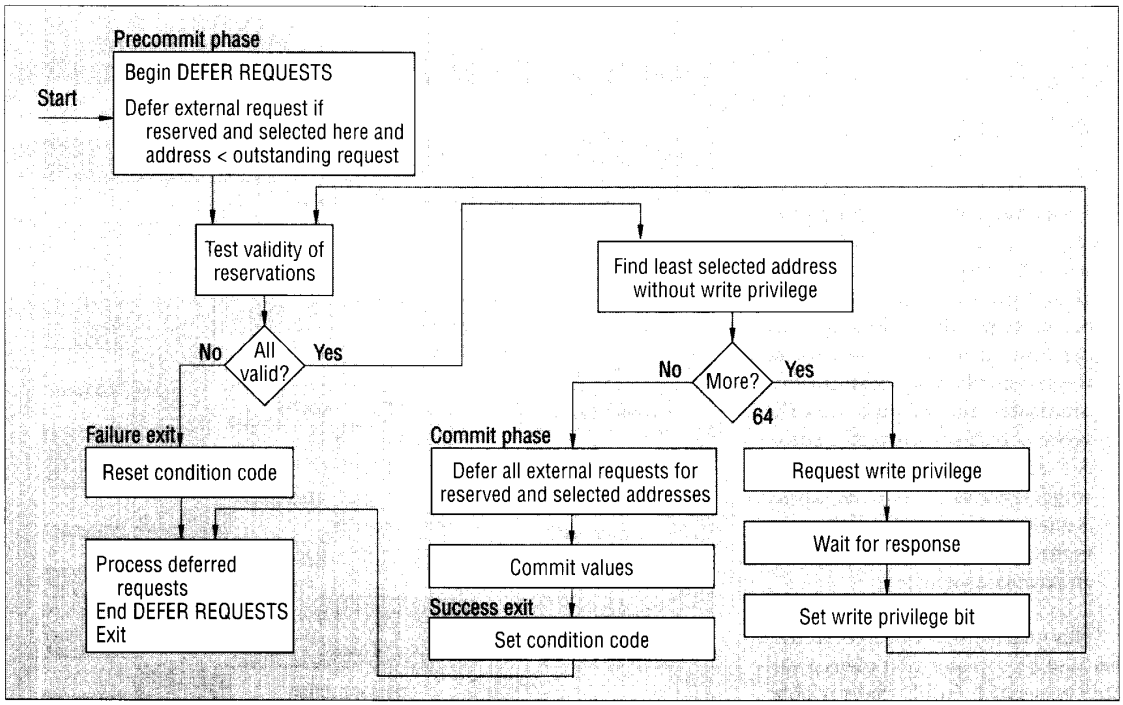

Figure 3. Execution logic for the Write-if-Reserved instruction.

can modify those variables until the process has ended. The instruction returns a condition code to indicate if the update has succeeded or not.

The implementation of Write-if-Reserved relies heavily on the underlying cache-coherence protocol, and can be added to a processor with such a protocol at relatively low cost.

The key to the implementation is to divide the Write-if-Reserved execution into two phases that mimic the two-phase locking protocols of transaction processing.[12,13] In the first phase, the processor requests write privilege for all specified reserved variables for which it does not already have write privilege. This uses the commands and replies available in the underlying cache-coherence protocol. When all requests have been answered, and the processor has obtained write privilege for all such reserved variables, it enters the second phase, which is equivalent to the *commit* phase of a database transaction. The processor holds write privilege for all specified reserved variables throughout this phase, preventing intervening writes by other processors. The commit phase must be executed as quickly as possible to minimize the performance impact of one processor holding exclusive control of a set of shared variables. It lasts long enough for the processor to modify the shared variables in its local cache and to send out the cache-coherence messages associated with these updates. The commit phase must be uninterruptible.

Because Write-if-Reserved grants exclusive rights to multiple resources, there is a possibility of deadlock. Two processors could try to update shared variables X and Y and reach deadlock when one owns X, the other owns Y, and both hold write privilege for one variable while waiting for the other to relinquish its write privilege for the other variable. To eliminate this possibility, Write-if-Reserved obtains write privilege in ascending order of address. This satisfies the sufficient condition of Coffman, Elphick, and Shoshani for deadlock freedom.[14] Write-if-Reserved performs the following steps during execution (see Figure 3):

(1) *Precommit phase:* Scan the specified reservation registers for validity and for reservations without write privilege.
(2) If all specified reservation registers are valid, then order the reservations without write privilege in ascending order of address, and issue requests for write privilege in that order. If any reservation becomes invalid before write privilege has been obtained for all specified reservations, then perform failure exit processing at step 5. During this period some requests for write privilege received from other processors can be deferred in a local buffer (described later).
(3) *Commit phase:* Update all modified shared variables. If requests for write privilege are received for any

reserved variables specified by Write-if-Reserved, defer them until the phase has completed.

(4) *Success exit:* Respond to all deferred requests for write privilege in the local buffer. Remove all specified reservations. Exit indicating that the update was successful.

(5) *Failure exit:* Respond to all deferred requests for write privilege in the local buffer. Remove all specified reservations. Exit indicating that the update was not performed.

Note that during the commit phase, write-privilege requests from other processors for specified reserved variables are deferred until the commit phase is over. If processor A in its commit phase modifies a variable for which processor B has requested write privilege from processor A, A first notifies B that the variable has changed value, and then grants write privilege. The first message invalidates a reservation at B if B holds one, and the invalidation occurs before B receives write privilege.

Multiple reservations shorten the effective length of the critical section in some codes, and they provide multiple concurrent updates when the updates do not conflict.

To prevent deadlock, special actions have to be taken during the precommit phase. When a processor receives a write-privilege request for a reserved variable specified by Write-if-Reserved for which it currently holds write privilege, it must do one of the following:

- If the address of the variable for which write privilege is requested is larger than the least reserved-address for which the processor does not have write privilege, grant the request. The processor releasing the reservation can either invalidate its reservation and abort the Write-if-Reserved, or continue and request write privilege again later. In the latter case, it is very likely that the variable will be modified before write privilege is returned, which will force the Write-if-Reserved to abort.
- If the address of the variable for which write privilege is requested is smaller than the least reserved-address for which the processor does not have write privilege, defer the request in a local buffer and honor it after the commit phase. If the request is deferred, the processor continues to hold a valid reservation for that address. Deferring the request may generate a reply message, if the implementation requires an immediate response to all requests.
- If the processor does not have write privilege for a requested address and therefore cannot grant write privilege, make a null response. This can be a negative acknowledgment message or no reply at all.

When processing deferred requests from the local buffer, a processor grants write privilege to the first deferred request for a particular variable; for subsequent deferred requests for the same variable, it sends messages that ask the requester to repeat the request because ownership has changed since the request was issued.

Multiple reservations are used in a context in which several variables must be updated concurrently by creating a *conditional sequence*:

(1) Issue a Read-and-Reserve for each shared variable to be updated and for each shared variable on which the update depends.

(2) Compute the new values of the modified shared variables. Save each modified value with a Store-Contingent instruction.

(3) For correctness, the success of a conditional sequence has to force the failure of any other conditional sequence whose set of reserved variables intersects with the set of variables reserved in step 1. Success is communicated by writing to one or more shared variables in the intersection of the reserved sets. Even if no variable in the intersection is modified, the code must have at least one Store-Contingent for a variable in the intersection. The Store-Contingent writes back the unchanged current value to the reservation register. The current value will be transmitted by a subsequent Write-if-Reserved, so no change of value in memory will occur, but the communication will invalidate reservations held elsewhere for the variable. (Such a Store-Contingent appears later in a list-management algorithm.)

(4) Execute Write-if-Reserved.

(5) Test the condition code returned by Write-if-Reserved. If the instruction was unsuccessful, return to step 1; otherwise continue.

Herlihy and Moss describe essentially the same set of instructions and a similar intent for their use, but they do not incorporate an equivalent deadlock avoidance into their implementation.[3]

Performance improvement

The major reason to provide multiple reservations is to remove critical sections from concurrent programs when possible. A critical section is a serial bottleneck. It limits the amount of useful concurrency available in a multiprocessor. For example, if a program has to execute N instructions in a critical section for every M instructions outside the critical section, then at most M/N copies of the program can execute concurrently on different processors, and speedup cannot exceed M/N. If more processors try to execute concurrently, the critical section will be occupied continuously, and the excess processors will be idle while awaiting entry to the critical section.

A disadvantage of a lock-free scheme based on reservations is that processors that fail in their attempts to perform updates of shared variables may expend a considerable number of cycles before discovering that the update has been unsuccessful.

Multiple reservations remove the effects of this bottleneck in at least two ways:

(1) They shorten the effective length of the critical section in some codes. Consider a code for updating a queue or a linked data structure. In codes with critical sections, we might need to embed all accesses to the queue or linked data structure so that at most one processor can access it at a time. In reservation-based code, the effective length of a critical section is the time between the first Read-and-Reserve and the Write-if-Reserved of a successful update sequence, which may be much less than the full length of the execution sequence that accesses the structure. This is true when reservation-based code tentatively protects different regions of the shared structure while searching for a region to alter. The effective length of a critical section includes only the code that protects the updated region, and excludes the code for regions that are temporarily reserved but not updated.

(2) They provide multiple concurrent updates when the updates do not conflict. In situations that involve incremental changes to linked data structures, different processors can perform local updates to different parts of the structure concurrently. In such cases, it can be impractical to provide concurrent multiple updates of the data structure by using critical-section methods. Hence, with critical sections, updates anywhere within a structure might have to be serialized.

Because atomic updates are done without placing locks on memory regions, a system that uses multiple reservations has a lower risk of system blockage due to a processor crash while it is updating shared variables. Assuming that the data are duplicated in memory or in another processor, a crash before or after the atomic update does not lead to total system failure because all other processors see either no update or the full update, depending on the point of the failure.

However, multiple reservations do not totally eliminate the possibility of system blockage. During the brief period, in the commit phase, when variables are written atomically to memory, the processor essentially holds a lock on them. A processor crash during this period could lead to system blockage, since the variables are inaccessible and might not become accessible again without external intervention. Depending on the underlying system assumptions, we might be able to incorporate fault tolerance by extending the cache-coherence protocol. One possible approach is based on a two-phase commit protocol[15] which is based on a write-update protocol, in which a modifying processor tentatively sends all the updates to other processors that have copies of the shared variables. It can also send them to a nonvolatile memory if this is required for fault tolerance by the base protocol. However, these updates are not applied until after a commit

message is sent by the processor performing the update. We do not have space here to fully discuss the implications of this fault-tolerance strategy.

Automatic restart

A disadvantage of a lock-free scheme based on reservations is that processors that fail in their attempts to perform updates of shared variables may expend a considerable number of cycles before discovering that the update has been unsuccessful. These cycles are lost because the update is not restarted at the earliest possible time. One way to avoid the loss of time due to delayed restart is to use a critical section in which processors wait at a lock. In this scheme, immediately after an atomic update completes, the lock opens, and a new process can initiate the next atomic update. Although this scheme is free of lost cycles due to delayed restart, it suffers from lost cycles due to waiting at the lock.

Since the Oklahoma update is wait-free, by incorporating a means for automatic restart, processors do not waste cycles unnecessarily because of delayed restart or waiting at locks. Thus, when multiple processors operate concurrently on a shared data structure using this scheme, all can do useful work when this is possible. For example, multiple processors can perform concurrent insertions and deletions in a linked list by using the algorithms in the next section. When the processors conflict, the conflict is detected and an automatic restart occurs. The wasted effort caused by the restart is no worse than the cycles lost waiting for a lock, but the potential to perform useful work when no conflicts occur offers increased performance over lock-based algorithms.

Since the Oklahoma update is wait-free, by incorporating a means for automatic restart, processors do not waste cycles unnecessarily because of delayed restart or waiting at locks.

Automatic restart can protect regions of memory whose contents change or are deallocated. All processors that modify a region reserve a shared pointer to that region. If processor A changes something in the region or deallocates the region's memory while processor B is scanning the region, then A invalidates the reservation, and B restarts immediately, before it can observe the changes or try to reference the freshly deallocated memory.

Automatic restart requires two additional fields in each reservation register (see Figure 1): *restart* contains the restart address for code that will begin the update process again, and *rvalid* contains a bit indicating (when set) that restart should occur automatically when the reservation is invalidated by another processor. (Bershad, Redell, and Ellis have described a more constrained automatic restart feature for uniprocessor systems, in which a process that is suspended during a restartable atomic sequence is automatically restarted at the beginning of the sequence.[16] Our technique can be used in that way as well.)

Automatic restart is controlled by a bit field in the Read-and-Reserve instruction. When this field contains a 0 bit, the instruction operates as described earlier. When it contains a 1 bit, the execution of this instruction places the instruction's address (or that of a restart point associated with the instruction) into the designated reservation register, and designates the address to be valid by setting the rvalid bit. The restart information stored in the reservation register should include or point to the values of general registers that are required to establish the context of the restart point.

Invalidating the reservation will force a restart at the address stored in the reservation register, if the address is active. The restart process can be handled as an interrupt, a conditional branch, or a subroutine call, depending on the implementation. The implementation has to assure that the restart is done in a correct context, so that the restart can run to a successful conclusion regardless of where the restart signal is received. If, for example, the signal is received when two levels of subroutine calls are active, the calls must be unwound two levels before restarting.

It is straightforward to force a restart from the beginning of the update process if any reservation is invalidated. Each Read-and-Reserve forces the address of the update's starting point to be saved in its reservation register. It is also straightforward to restart the process at the Read-and-Reserve instruction whose reservation has been lost, and to repeat only the code from that point. This eliminates the repetition of the code prior to that point.

It is less straightforward, but still possible, to reexe-

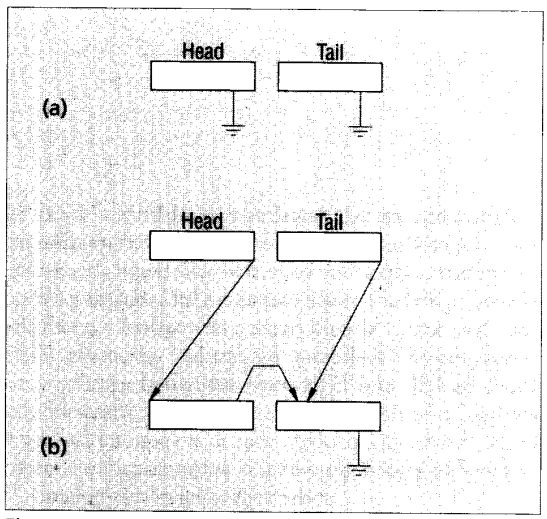

Figure 4. Typical queue structure: (a) an empty queue; (b) a nonempty queue.

cute only those instructions that need to be repeated. The details depend on the exact nature of the code, and can become complicated. The hardware support must enable restart at a point that depends on which reservation was invalidated, and must allow the machine context to be restored correctly when the restart initiates.

Livelock avoidance

The Read-and-Reserve, Store-Contingent, and Write-if-Reserved instructions guarantee that the system as a whole will make progress when multiple tasks contend for shared variables. If concurrent processors use intersecting sets of reserved variables that are accessed only with these special instructions, and if several processors execute a Write-if-Reserved, one processor succeeds. However, because the intended implementation is based on cache coherence, some variables may be shared inadvertently because they lie in the same cache line. This *false sharing* may cause a reservation to be canceled by a store made by another processor to any location in the same cache line as a reserved variable. Two processors may repeatedly cancel each other's reservations by an ordinary store to a variable that shares a cache line with a reserved variable, with no processor able to make progress.

This problem occurs for single-reservation systems as well as for our proposed multiple-reservation system. One recommended solution is to allocate shared variables dynamically in ways that eliminate false sharing. An alternative is not to perform any ordinary stores in a conditional sequence.

However, we suggest an alternative that allows arbitrary conditional sequences to contend without risking this kind of livelock. The central notion is a well-known exponential-back-off technique.[17,18] If a processor executes a conditional sequence unsuccessfully and elects to retry it, the processor delays before its next attempt; each successive attempt has a delay that is double the delay of the previous attempt. If this delaying technique is followed by all N processors contending to modify a shared variable in a conditional sequence that can be performed within M cycles, then one processor will successfully modify the shared variable within $N \times M$ cycles. Thus, the back-off mechanism assures that at least one processor makes progress. However, it does not prevent an individual processor from experiencing starvation. While it is improbable and unusual for a processor to fail repeatedly when it executes Write-if-Reserved using the back-off mechanism, it is not impossible.

The hardware implementation of exponential back-off associates a delay with a restart address. The *delay* field in Figure 1 is a counter that can hold a value up to $N \times M$, where N is the maximum number of processors that may contend, and M is a bound on the cycles a conditional sequence can take. The counter is given an initial value of 1 when the Read-and-Reserve is first executed, and is shifted left for each subsequent restart, but no shift occurs when the 1-bit reaches the most significant position in the field.

When a restart occurs, it is initiated after a variable delay that depends on the contents of the delay field. The field's binary value is an integer 2^i. The length of the delay before restarting is an integral number of clock periods t chosen at random from the interval $2^i \leq t < 2^{i+1}$. This distributes the restart times so that some processor can begin early enough to complete successfully in the next interval, while other processors delay long enough to avoid interference.

This extension to the special instructions protects against livelock induced by false sharing and supports a guarantee of overall progress. In any particular system, other implementation details, such as the treatment of reservations during context swaps and interrupts, might affect the validity of such a progress guarantee.

Sample programs

An important use of multiple reservations is for managing queues. Figure 4 shows a typical queue structure in which a queue is accessed through two pointers, Tail and Head. Figure 4a shows an empty queue, and Figure 4b shows a queue with two items. Items in the queue are linked by one-way pointers from the front (Head) to the rear (Tail).

The Enqueue procedure adds a new item to the rear of the queue, and the Dequeue procedure removes items from the front. We could create Enqueue and Dequeue procedures that use an architecture with a single reservation register; however, because the code might have to change multiple shared variables in some cases, it must protect against inconsistencies that could arise when the variables are changed nonatomically.

An architecture with multiple reservations enables the changes to be made atomically. It is simple and efficient code. Moreover, it is sufficiently brief to be used in-line, eliminating the overhead of a subroutine call and streamlining the fetching and paging of code.

Figure 5 shows the Enqueue procedure. The *if* statement handles the queue's two possible states: empty and nonempty. (The pseudocode in the figure omits *begin/end* pairs and similar bracketing. Instead, it uses indentation to indicate which statements fall within an *if*, *else*, or *repeat* statement.)

- Read-and-Reserve(Memory[*X*].*next*, *Y*) loads the address of the field *next* of location *X* into the address field of reservation register *Y*. The data in the *next* field is copied into a general register. Any modification of the *next* field at *X* by another processor invalidates the reservation.
- Store-Contingent(*X*, *Y*) places value *X* in the data field of reservation register *Y* in preparation for an Oklahoma update.
- Write-if-Reserved(*X*, *Y*) does an Oklahoma update on the reservation registers *X* and *Y*. In the figure, the update is performed on the *next* fields of the reserved addresses.

Figure 6 shows the Dequeue procedure, which must deal with three cases: an empty queue, a queue with one item (the queue becomes empty), and a

```
Procedure Enqueue(newpointer)

Memory[newpointer].next := nil
status := "unfinished"

repeat

   last_pointer := Read-and-Reserve(Memory[tail].next, reservation1)

   if last_pointer = nil then
      {empty queue}
      first_pointer := Read-and-Reserve(Memory[head].next, reservation2)
      Store-Contingent(newpointer, reservation1)   {Updated tail}
      Store-Contingent(newpointer, reservation2)   {Updated head}
      status := Write-if-Reserved(reservation1, reservation2)

   else
      {one or more items in queue}
      temp_pointer := Read-and-Reserve(Memory[last_pointer].next, reservation2)
      Store-Contingent(newpointer, reservation1)   {Updated tail}
      Store-Contingent(newpointer, reservation2)   {Updated pointer in former last
                                                    item in queue}
      status := Write-if-Reserved(reservation1, reservation2)

until status is "successful"
```

Figure 5. The Enqueue procedure.

```
Procedure Dequeue(first_pointer)

status := "unfinished"

repeat

   first_pointer := Read-and-Reserve(Memory[head].next, reservation1)
   if first_pointer = nil then
      {empty queue}
      Write-if-Reserved(reservation1)              {Remove the reservation}
      status := "successful"

   else
      last_pointer := Read-and-Reserve(Memory[tail].next, reservation2)
      if first_pointer = last_pointer then
         {one-item queue}
         Store-Contingent(nil, reservation1)       {Updated head}
         Store-Contingent(nil, reservation2)       {Updated tail}
         status := Write-if-Reserved(reservation1, reservation2)

      else
         {two or more items in queue}
         second_pointer := Memory[first_pointer].next
         Store-Contingent(second_pointer, reservation1)   {Updated head}
         status := Write-if-Reserved(reservation1, reservation2)  {Clear both
                                                                   reservations}

until status is "successful"

return (first_pointer)
```

Figure 6. The Dequeue procedure.

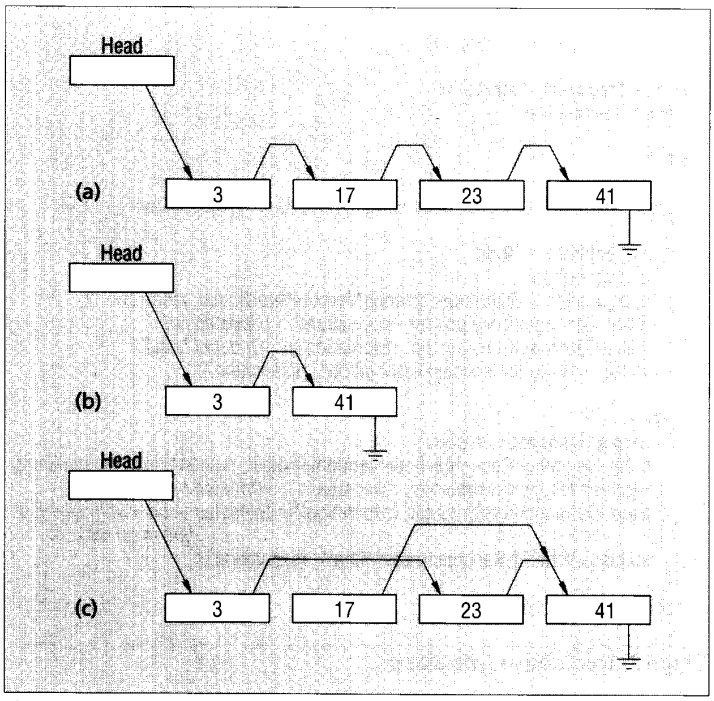

Figure 7. A one-way linked list prior to deletion (a), after the second and third items have been deleted (b), and after two concurrent deletions have left it in an incorrect state (c).

queue with multiple items (the queue remains nonempty). There can be concurrent attempts to enqueue and dequeue on any of these three queue states; in all cases, the actions are correct and leave the queue in a consistent state.

List insertion and deletion

Let's now look at a one-way linked list, in which arbitrary items can be deleted, to show that two or more deletes can be done concurrently and consistently without locking when two reservation registers are available.

Figure 7a shows a one-way linked list before a deletion. The list has four elements and a reserved cell (Head) that points to the first element. Even though only one pointer changes at a time to perform a deletion, two pointers must be reserved to assure consistency. To see why this is true, assume that one processor tries to delete item 17, and another tries to delete item 23. Figure 7b shows the form the list would take if the deletions were serialized. If both processors perform their deletions in the same cycle, and both have reserved only the link that they are about to change, the deletions can occur concurrently, and the list will be in an erroneous final state such as that shown in Figure 7c, where item 17 has been deleted by pointing item 3 to item 23, and item 23 has been deleted by pointing item 17 to item 41. This final state is inconsistent with the intent of the deletions. The correct result is a list containing only items 3 and 41, whereas the list actually contains items 3, 23, and 41. This failure occurs because the processor that was deleting the third item moved a pointer in the second item, which itself was being deleted.

A correct program must use two reservations, reserving not only the pointer to be changed but also the pointer to the cell that contains the changed pointer. This cell must remain reachable throughout the atomic update, so the pointer to it must not change while the pointer contained within the cell changes. Figure 8 shows a program that deletes an item from the list. It detects the unreachability of a cell that contains a pointer to be changed by reserving the pointer to that cell. The variable *reservation*2 reserves the cell to be altered, while *reservation*1 reserves a pointer to that cell. If the cell to be altered is being deleted and deallocated by another processor, that processor must alter the pointer reserved by *reservation*1. By reserving both cells, the algorithm guarantees that the cell is changed correctly, and that the cell itself has not been deleted from the list by a different processor executing a conditional sequence.

The program in Figure 8 also illustrates the use of automatic restart to protect codes from accessing unreachable regions of memory. The pseudocode notation is similar to that in Figures 5 and 6, except that Read-and-Reserve also indicates a restart address, which appears as an optional argument giving a label. No additional context information has to be saved because there is no context change within the program.

Each item has two fields: *contents* and *next*. The *contents* field gives the actual data for the item. The *next* field is a pointer to the next item on the list and has the value *nil* if there is no next item. The program searches through a linked list for a match with the input parameter *delete_point*. If the program reaches the end of the list, indicated by a *nil* pointer value for the variable *succ*, the program exits without making a deletion after clearing the reservations that have been placed.

For this data structure, the empty list is denoted by a *nil* pointer value in the *next* field of the head. Because the head cannot be deleted, the program uses only a sin-

```
Procedure Delete(delete_point)

Init_pred: pred := Read-and-Reserve(Memory[head].next, reservation1, recover: Init_pred)

if pred = nil then
  Write-if-Reserved(reservation1)  {Clear the reservation}

else
  if Memory[pred].contents = delete_point then
    new_head := Memory[pred].next
    Store-Contingent(new_head, reservation1) {Update head}
    Write-if-Reserved(reservation1)

  else
    Init_succ: succ := Read-and-Reserve(Memory[pred].next, reservation2, recover: Init_succ)

    while (succ ≠ nil) and (Memory[succ].contents ≠ delete_point) do
      pred := Read-and-Reserve(Memory[pred].next, reservation1, recover: Init_pred)
    Reset_succ: succ := Read-and-Reserve(Memory[pred].next, reservation2, recover: Reset_succ)

    if succ ≠ nil then
      {Delete-point has been reached}
      temp_pointer := Memory[succ].next      {Pointer to element after delete point}
      Store-Contingent(pred, reservation1)    {Write back old value of pred.next}
      Store-Contingent(temp_pointer, reservation2)
      Write-if-Reserved(reservation1, reservation2)

    else
      {No match, therefore no delete. Just clear the reservations.}
      Write-if-Reserved(reservation1, reservation2)
```

Figure 8. Delete procedure for a one-way linked list.

gle reservation while deleting the first list element. For all other deletions, *succ* is the pointer changed to perform the delete, and *pred* is the pointer through which *succ* is linked to the list. By reserving *pred* during the update, the program guarantees that *succ* is not deleted while being updated.

Because of automatic restart, the program does not have to reach the Write-if-Reserved instruction to detect that a restart is required. The restart becomes effective as soon as another processor changes a reserved pointer.

Note that the program updates both pointers when it executes Write-if-Reserved. The value of *pred* is unchanged by this update, but the location is still updated to force a restart by other processors that have reservations on this pointer.

Immediately after executing the statement prior to the label *Reset_succ*, the program can reach a state in which both reservation registers reserve the same address. If that address is modified at this instant, there could be a problem in restarting, since two different restart addresses can be invoked. In this code, the restart address in the reservation register with the lowest index has priority. This forces the restart to occur at *Init_pred* rather than at *Reset_succ*.

Overhead is negligible in this implementation because there are no locks to manage. The worst-case behavior occurs when contention is high and restarts are frequent. But some tasks are guaranteed to complete their work because it takes a successful Write-if-Reserved to invalidate other reservations. Hence, frequent restarts are accompanied by frequent successful executions of Write-if-Reserved.

A companion program for list insertion forces an update to *reservation*1 when updating *reservation*2 in case the cell protected by *reservation*2 is being deleted by another processor (see Figure 9).

To verify correct concurrent operation of the insert and delete programs, we need only confirm that two inserts can run correctly when they insert at the same or adjacent places on the list, that two deletes can attempt to delete the same or adjacent items in either order, and that a delete and an insert can work correctly when they operate on the same or adjacent cells.

In these examples, two reservations are enough to manage a FIFO queue and a linked list with insertions and deletions at arbitrary points. A doubly linked list can be implemented with four reservations. While there is no bound on the number of reservations required for an arbitrary transaction, it is clear that three or four reservations suffice for a large class of commonly used functions.

```
Procedure Insert(insert_point, new_item)

Init: firstp := Read-and-Reserve(Memory[head].next, reservation1, recover: Init)

 if (firstp=nil)
    Memory[new_item].next := nil;
    Store-Contingent(new_item, reservation1) {Update head to point to new_item}
    Write-if-Reserved(reservation1)

 else
    Init_secp: secondp := Read-and-Reserve(Memory[firstp].next, reservation2, recover: Init_secp)
    while ( (secondp ≠ nil) and (Memory[secondp].contents ≠ insert_point) ) do
       firstp := Read-and-Reserve(Memory[firstp].next, reservation1, recover: Init)
       Reset_secp: secondp := Read-and-Reserve(Memory[firstp].next, reservation2, recover: Reset_secp)

    {Insert-point has been reached}

    Memory[new_item].next := secondp          {Insert new item before secondp}
    Store-Contingent(firstp, reservation1)    {Dummy update of pointer to the cell whose pointer is modified}
    Store-Contingent(new_item, reservation2)  {Update a pointer to point to new_item}
    Write-if-Reserved(reservation1, reservation2)
```

Figure 9. Insert procedure for a one-way linked list.

Advantages of multiple reservations

If the list-insertion and deletion programs used only a single critical section for insert and another for delete, then only one processor at a time could do either an insert or a delete. This would hurt performance by creating a long critical section and by limiting concurrency during its execution. If the programs were structured so that each loop iteration were embedded within one critical section, the length of the critical section would be much shorter, but no two programs could traverse the same part of the linked list concurrently. This would tend to hold processors at early points in the list and keep them from examining later portions where concurrent updates might be performed. With multiple reservations, however, there are no restrictions on the number of processors that can search a list concurrently or where the search must be conducted. The effective parallelism is limited to the number of nonconflicting updates that can be executed concurrently.

The Oklahoma update provides an evolution path for developing concurrent algorithms that closely resemble their trusted serial versions, and are therefore easily created and verified. The method also provides an evolution path toward highly parallel systems and continuously available systems. In both systems, the guarantee of forward progress is an essential feature for management of shared variables. ▨

Acknowledgment

The authors gratefully acknowledge the helpful comments of Mark Charney and the referees in the preparation of the article.

References

1. H.S. Stone, *High-Performance Computer Architecture*, third edition, Addison-Wesley, Reading, Mass., 1993.
2. M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures," *Proc. Second ACM SIGPlan Symp. Principles and Practice of Parallel Programming, SIGPlan Notices*, Vol. 25, No. 3, Mar. 1990, pp. 197-206.
3. M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. 20th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 289-300.
4. B. Lampson, "Atomic Transactions," in *Distributed Systems – Architecture and Implementation, Lecture Notes in Computer Science 105*, B.W. Lampson, M. Paul, and H.J. Siegert, eds., Springer-Verlag, Berlin, 1981, pp. 246-265.
5. M. Banâtre and P. Joubert, "Cache Management in a Tightly Coupled Fault-Tolerant Multiprocessor," *Proc. 20th Fault-Tolerant Computing Systems Symp.*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 89-96.
6. R.K. Treiber, "Systems Programming: Coping with Parallelism," IBM Research Report RJ 5118, IBM T.J. Watson Research Center, 1986.
7. E.H. Jensen, G.W. Hagensen, and J.M. Broughton, "A New Approach to Exclusive Data Access in Shared-Memory Multiprocessors," Tech. Report UCRL-97663, Lawrence Livermore Nat'l Lab., 1987.
8. G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J., 1992.
9. R. Sites, ed., *DEC Alpha Architecture*, Digital Press, Burlington, Mass., 1992.
10. IBM, *The IBM Power PC Architecture – A New Family of RISC Processors*, Morgan Kaufmann, San Mateo, Calif., 1994.
11. P. Sweazey and A.J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus

Cache Management in a Tightly Coupled Fault-Tolerant Multiprocessor," *Proc. 13th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1986, pp. 414-423.

12. K.P. Eswaran et al., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM*, Vol. 19, No. 11, 1976, pp. 624-633.

13. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.

14. E.G. Coffman, Jr., M.J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys*, Vol. 3, No. 1, 1971, pp. 67-78.

15. J.N. Gray, "Notes on Database Operating Systems," *Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60*, Springer-Verlag, Berlin, 1978, pp. 393-481.

16. B.N. Bershad, D.D. Redell, and J.R. Ellis, "Fast Mutual Exclusion for Uniprocessors," *Proc. ASPLOS-V*, ACM Press, New York, 1992, pp. 223-233.

17. *IEEE Standard for Local Area Networks, 802.3: CSMA/CD Access Method*, IEEE Press, Piscataway, N.J., 1985.

18. T.E. Anderson, "The Performance of Spin-Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 1, Jan. 1990, pp. 6-16.

Janice M. Stone is an advisory programmer at the IBM T.J. Watson Research Center. Her research interests focus on parallel algorithms and tools for developing and analyzing them. Her recent work investigates the PowerPC's shared-memory architecture. She received her BA in mathematics from Duke University in 1962, and she has pursued graduate studies in mathematics at Georgetown University, and in logic and the philosophy of science at Stanford University.

Harold S. Stone is a research staff member at the IBM T.J. Watson Research Center. He has been a faculty member at the University of Massachusetts and Stanford University, and he has held visiting faculty appointments at the University of California at Berkeley, New York University's Courant Institute of Mathematical Science, Cornell University, and others. He holds seven patents, has written several textbooks and more than 70 technical publications, and has been on the editorial boards of *Computer*, *Transactions on Parallel and Distributed Systems*, and the *Journal of the ACM*. He has also been a member of the IEEE Computer Society's Board of Governors. Stone received his PhD in electrical engineering from University of California at Berkeley. He is a fellow of the IEEE. He received the Charles Babbage Outstanding Scientist Award in 1991, and the IEEE Piore Award in 1992.

Philip Heidelberger is a research staff member at the IBM T.J. Watson Research Center. His research interests include the modeling and analysis of computer performance, probabilistic aspects of discrete event simulations, and parallel simulation. He is a member of the editorial board of ACM's *Transactions on Modeling and Computer Simulation*, and has been a board member of *Operations Research*. He received his PhD in operations research from Stanford University in 1978, and his BA in mathematics from Oberlin College in 1974.

John Turek is a research staff member at the IBM T.J. Watson Research Center. His research interests include database systems, distributed computing, and optimization problems in computer science. He received his PhD and MS degrees from New York University's Courant Institute of Mathematical Science in 1991 and 1990, respectively, and his BS degree from MIT in 1984.

The authors can be reached at the IBM T.J. Watson Research Center, Yorktown Heights, NY 10598; Internet jmstone@watson.ibm.com