Optimistic Synchronization in Distributed Shared Memory*

Gudjon Hermannsson

Renaissance Technologies Corp.
Stony Brook, NY 11790

Larry Wittie

Computer Science Department, SUNY
Stony Brook, NY 11794-4400

Abstract

This paper introduces optimistic lock synchronization using the group write consistency model (GWC). GWC guarantees strict ordering of all shared writes in a processor group. In optimistic synchronization, if a lock-requesting processor can assume that the lock is free, execution of mutually exclusive code starts immediately. Wrong assumption results in rollback. Shared variable updates remain in the group until the lock manager grants the lock to the requesting processor.

By evaluating the time needed for three processors to execute mutually exclusive code, GWC can outperform weak, release, and even entry consistency. Simulations of task management using exclusive access to a shared queue, also show much faster mutual exclusion with GWC. Optimistic mutual exclusion may further halve total delays in accessing shared resources.

1 Introduction

Distributed shared memory(DSM) systems are networks of computers that transparently share variable values among processors. They are message passing systems, but hide the underlying message passing mechanism from programmers and allow a shared memory programming paradigm. DSM systems keep the conceptualization advantages of shared memory and yet have the potential for scaling efficiently to highly parallel systems with thousands of processors.

Remote memory accesses and synchronizations are the two major activities, besides load imbalance, that avoidably lengthen execution times for parallel programs running on DSM machines. Factors that determine the length of delays are: how the system handles remote memory accesses to the logically shared data space, the consistency model employed to relax constraints on memory access order, and the synchronization methods available.

Synchronization primitives make programs easier to write and understand, but processors waiting for locks are wasting time. Synchronization is used in nearly every parallel program. Lessening lock delays is a major goal for efficient execution of parallel programs.

This paper introduces optimistic mutual exclusion. This new synchronization method allows safe execution of lock-protected code before lock permission is granted. In the best case, useful computations totally overlap lock confirmation, halving the total time for synchronization plus exclusive execution. If another processor gets the lock, a simple roll back is performed.

Optimistic synchronization is based upon group write consistency(GWC)[11] and eagersharing of remote memory changes[18]. This paper shows how optimistic synchronization uses local memory copies to estimate global lock state and how to ameliorate long lock delays in huge networks, possibly producing zero waste mutual exclusion.

1.1 Methods to Access Remote Memory

Remote access mechanisms for logically shared distributed memory form a spectrum. At one end are demand-driven methods, which delay accesses to remote data until each is actually needed, but the processor must halt until each remote datum can be fetched. Network traffic is minimized. At the other end are eagersharing methods built on the principle that as soon as a shared datum changes, it is sent over the network to all processors that may need it. The main goal of eagersharing is to have remote data already present in local memory whenever needed. Ideally, useful computations overlap all communication delays, and processors are never idled.

Demand-fetch protocols do not scale well; for many important parallel algorithms, they do not execute effi-

*This research has been supported in part by Department of Energy/Superconducting Super Collider contract SSC-91W09964; by National Science Foundation grants for equipment (CDA88-22721, CDA90-22388, and CDA93-03181) and research (MIP89-22353); and by Office of Naval Research grants N00014-88-K-0383 and N00014-93-I-0625.

1063-6927/94 $03.00 © 1994 IEEE

ciently on more than a few dozen processors[18]. Most applications can be structured so that much less than 3% of all memory references are shared writes[4]. Eagersharing of writes allows efficient execution in much larger networks than does demand-fetch access.

Eagersharing and cache update are similar, but dynamic disabling of eagersharing can avoid some costs. Eagersharing is roughly equivalent to cache update combined with prefetch of soon-to-be-needed variables and with flushing of no longer needed cache lines.

1.2 Parallel Computing Data Consistency

Consistency models place specific requirements on the order in which shared memory accesses from one processor may be observed by other processors in a multiprocessor system [6, 15]. To design correctly functioning, efficient programs, a programmer of a parallel computer must know its consistency model.

The strictest model is sequential consistency[13], which requires both read and write memory accesses to appear on all computers in the same order, as if all process executions were somehow interleaved on a sequential machine. It is inefficient even for two processors. A weaker model is processor consistency[8], which allows reads to bypass pending writes to gain efficiency. Total store ordering[12] ensures the same order for all writes to memory. However, its use of a centralized memory write arbitrator is not viable for large distributed memories. Partial store ordering[12] allows any order of writes between explicit storage synchronization markers, but completes all writes in one marker section before the next starts.

Weak consistency[3] is guaranteed consistency only at synchronization points. Release consistency[6] gains efficiency by using knowledge of the type of lock access to allow more pipelining. Entry consistency[2] goes further by associating dataguards with data and requiring consistency only when entering a guarded data section. For most models, memory accesses may be performed out of order between synchronization points, but data must be consistent on all processors at those special points.

Group write consistency(GWC)[11], provided by hardware interfaces for Sesame (Scalable Eagerly ShAred MEmory) networks, is efficient for massively parallel computers. It produces total store ordering within each small group of processors. It gives more precisely structured data changes than processor consistency, but is much faster. Processor groups overcome the total store ordering arbitration bottleneck.

A reliable tree-based multicast protocol is implemented in hardware by the memory sharing interfaces that link Sesame workstations[18]. One processor that writes to the variable is root for the spanning tree used to route, to sequence, and to retransmit all hidden sharing messages within the group.

Group write consistency guarantees the order of writes within each sharing group whether the writes are from one source or many. Group write consistency could also guarantee ordering between overlapping groups, as is done in an independently developed, similar distributed algorithm for ordering messages[7]. However, for many coding applications, complete ordering is not needed. Combining overlapping groups into one global group can prevent scaling in large networks by overloading the global root and greatly reducing performance. In Sesame, explicit mutual exclusion can enforce ordering for overlapping groups in the rare cases when it is needed.

One major difference distinguishes group write consistency from other models. Others force each processor to wait at synchronization points or possibly at every remote memory access until the previous write has changed memory on all other processors before continuing. Instead, Sesame hardware[18] guarantees the consistent local ordering of writes. This difference is extremely important in networks of thousands of computers. All eagerly shared writes are intercepted by memory sharing hardware and will be performed in the same order on all sharing processors. For each synchronized write, a computer using older consistency models encounters the delays of a round-trip. Using the write ordering of group write consistency, a processor can immediately perform the next instruction, even if it is another shared write.

The write ordering of group write consistency reduces synchronization delays since ordinary shared variables can have special meanings, for example, can be reader-writer locks distributed with shared data structures to eliminate most synchronization penalties when there is only one writer. This paper also shows that mutual exclusion under Sesame's combination of group write consistency and eagersharing gives greater efficiency than with other models. Lock grant and release synchronization accesses can safely accompany the last shared write in a mutual exclusion section. As this paper shows, optimistic synchronization attempts to overlap all communication delays with execution of the mutual exclusion section.

1.3 Synchronization

Hardware primitives for repeated lock tests such as Test-and-set[3], Test-test-and-set[17], and their extensions[1] evolved on shared memory multiproces-

sors. In distributed systems repeatedly testing locks produces too much network traffic. Queue-based locks[9, 10, 1, 5, 14, 2, 16] are alternatives. A lock request is sent to a lock owner. If the lock is free, permission is granted. If it is busy, the request is queued. When the lock becomes available, the next process in the queue gets permission. Lock queues can be supported in hardware[9, 5] or software[10, 1, 14, 2, 16]. Queue-based locks are needed in distributed memory systems, even those with local lock copies, to lessen network traffic after lock release.

When moving from multiprocessors connected by busses to multicomputers connected by networks, locating the lock owner becomes an issue. Distributed directory schemes[5] allow a lock request to go directly to the lock manager or through the lock manager to the current owner. Other schemes[2, 16] use a distributed algorithm to guess the current lock owner, p. If the guess is wrong, there are two possibilities: p is waiting for the lock and the request is queued at p, or the request is forwarded to a new guess supplied by p. Group write consistency uses queue-based locks and a group root as a lock manager. In almost all cases, the lock manager is the lock owner and lock requests will never have to travel further.

Section 2 is an explanation of the way an eagerly shared variable can be used for synchronization under group write consistency. Section 3 includes comparisons of idle times during non-optimistic mutual exclusion for group write consistency versus release consistency. Section 4 is a description of optimistic synchronization under group write consistency.

2 Group Write Consistency Locks

Since writes are ordered, the case for one writer is simple; an ordinary variable can lock a data structure awaited by reader(s). If code on the writing processor finishes all data updates before unlocking the variable, all processors will see the same order of changes. Each processor can check its local lock to see whether the data is valid. Relocking while data is being read can trigger rereading to get consistent data values.

The ordering of group write consistency allows a new, very efficient mutual exclusion algorithm for eagersharing systems. Compiler tools can aggregate related variables and locks into the same sharing group. Each lock is initially set to a unique negative number not matching any positive processor number, say -99..99, meaning **free**. When the variable is positive, the processor with that unique identification (ID) number has exclusive access.

A processor wanting exclusive access writes the lock variable with the negated value of its own processor number. The write is copied by the local eagersharing monitor and sent to the group root. The root checks if the lock is **free**. If not **free**, the processor ID number is queued. If **free**, the root writes the positive processor ID into the lock variable to grant permission. When the original node sees its own positive ID arrive in the lock value, it can continue execution. As each processor frees the lock by writing -99..99 in its local copy, the root checks whether any nodes are queued awaiting exclusive access. If so, the next queued number is written as the new lock value. If not, the **free** value (-99..99) is propagated to all group memories. A processor always receives exclusive access within one or one half round-trip time of the lock being freed. There is no network traffic except three one-way messages to request, grant, and release the lock.

GWC has an advantage for heavily requested locks. The last exclusive write is followed by local lock release. The group root can immediately append the next lock grant to the shared data written by the previous processor. On each node, the writes complete before the lock changes. Mutual exclusion across multiple groups requires permissions from all the involved roots. Routing corresponding locking messages and data changes on the same paths through the roots guarantees a consistent view of variable updates.

3 Non-Optimistic Mutex Comparison

Figure 1 compares wasted idle times for three successive sets of mutually exclusive accesses under Sesame group write, entry[2], weak[3], and release consistency[6]. Each part show times for contending requests to the same lock. Eager sharing or cache update sharing of data are used to minimize data access delays in all cases except entry consistency, which has the policy of not updating until a lock is requested. Weak and release consistency behave the same since each processor locks, reads or updates, and releases only once. In all models, only one processor at a time is allowed to access the locked data, even though copies may be present on the other processors. $CPU2$ requests exclusive access later than $CPU1$ and $CPU3$.

In Figure 1(a) for GWC, the Sesame interface for $CPU1$ copies its local data changes without slowing its calculations. The dashed lines indicate shared data being sent to the group root, and the solid lines data from the root to all destinations. When $CPU1$ finishes its last update, it immediately releases the lock. The shared writes reach group root $CPU2$ and are

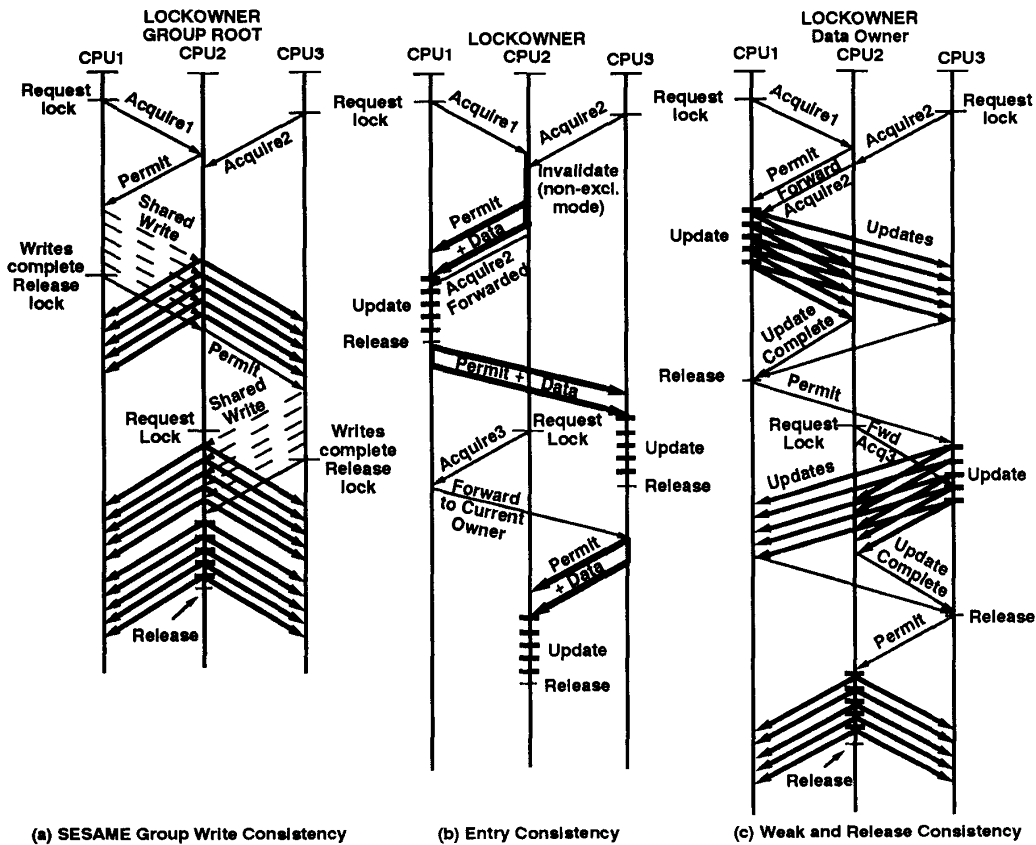

Figure 1: Locking Comparison

redistributed via its tree to all (three) sharing CPUs. The lock release reaches $CPU2$ after the last written datum, as guaranteed by write order. The release immediately becomes a permission forwarded to queued processor $CPU3$. When $CPU3$ receives the lock permission, it can read the shared data locally and perform its own updates. After the last update, the lock release is sent back to $CPU2$ for its updates. Sesame is efficient even under heavy contention. It is not possible to release the lock sooner than the last datum update, nor to get permission before the last changes to shared data reach the requesting processor.

Lock requests under entry consistency are shown in Figure 1(b). Again two CPUs request the lock. Locks under entry consistency can be requested in either non-exclusive mode or exclusive mode, and invalidation is used to transfer from non-exclusive to exclusive mode. Before $CPU1$ is given permission, the lock owner send an invalidation to the processors holding the data in non-exclusive mode. Data changes and lock permission are sent to $CPU1$. The request by $CPU3$ is forwarded to $CPU1$ and queued. When $CPU1$ finishes, it releases the lock, and both the changed data and lock permit are sent to $CPU3$. After $CPU3$ gets permission, it writes the shared data and releases the lock locally. When $CPU2$ requests the lock, it sends the lock request to its best guess for the lock owner, $CPU1$, which forwards the lock request to $CPU3$.

Similar lock requests under weak and release consistency are shown in Figure 1(c). Again two CPUs request the lock, $CPU1$ is given permission, and the request by $CPU3$ is forwarded to $CPU1$ and queued. This method may need three one-way messages to get a lock [5], for example, the request by $CPU3$ to the lock manager ($CPU2$) is forwarded to the current lock owner ($CPU1$), which must eventually grant the lock to $CPU3$. When $CPU1$ finishes, lock release to $CPU3$ is blocked until the updates reach all nodes.

The same time scale is used in all three parts of

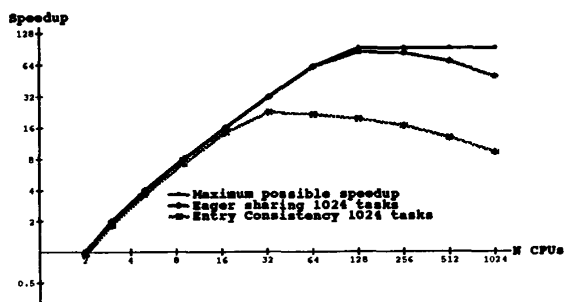

Figure 2: Speedup for Task Management: $\frac{1}{100}$

Figure 1, which shows that Sesame GWC is better than entry, weak, or release consistency, for this example. Entry consistency is not as rapid as Sesame. A round-trip invalidation is needed to move data in non-exclusive mode to exclusive mode. If several processors are contending heavily to acquire the lock, entry consistency performs as well as possible for Sesame, except for the extra data transmission time needed after each local lock release. GWC avoids this extra propagation delay. Under light contention, entry consistency may not perform as well, since a new requestor may often guess the wrong lock owner and have to wait for its request to be forwarded.

Weak and release consistency take much longer than GWC, and take even longer if invalidation is used. For very large systems, the disparity between group write consistency and the other models will be significantly larger than shown, since network delays will be much longer than local update times.

3.1 Simulation Results

This section shows system speedups for a task management application that uses eager sharing combined with GWC for mutual exclusion. Eager sharing is compared to a fast version of entry consistency, which is assumed always to know the lock owner, so no time is ever lost in relaying requests to find the lock owner. All releases in entry consistency are local.

Figure 2 illustrates effective network power when one producer generates a total of 1024 tasks and waits for the last to be executed before stopping. The top line shows the maximum speedup possible if network delays were zero. Speedup is average processor efficiency times network size. Efficiency is the percentage of peak processor speed. The time to produce a task is assumed to be shorter ($\frac{1}{100}$) than the time to process a task. Since the time to generate 1024 tasks is negligible compared to the execution time, the producer is effectively an idle processor. For 2 processors, minutely more than 50% is the maximum efficiency, resulting in an effective speedup of 1. The number of processors in Figure 2 is a power of two plus one $(3, 5, 9, ...)$ to eliminate load balancing effects. As can be seen in Figure 2, both the extra time for entry consistency to send the changed data with the lock and the waits for updated read copies of values protected by a lock become significant for larger networks.

Sesame reaches a peak speedup of 84.1 from 129 processors. Mutual exclusion request delays are more visible for larger networks. The time ratio assumption of $\frac{1}{100}$ for task production versus task execution causes the drop in efficiencies for large networks. With over 100 processors, there are not have enough tasks produced to keep all processors busy. A processor finishing a task has to wait to get a new task to execute. For entry consistency, peak speedup is only 22.5 from 33 processors. GWC gives 3.7 times faster performance.

Eager sharing allows much more efficient locked queue management than entry consistency. For entry consistency, the maximum possible network power is never reached if there are more than 2 processors. For more processors, the gap between the realized and maximum attainable speedup widens slowly up to 17 processors and very rapidly after that. Entry consistency provides less speedup than Sesame GWC for two reasons: it takes extra time to send the data just before each lock, and the processors must fetch and test a variable written by the producer to check if the processor queue is full, causing network traffic and delays. With eager sharing, the test variable is immediately sent to all processors whenever it changes.

4 Optimistic Mutual Exclusion

This section presents a new algorithm to hide all or part of the network round-trip delay to request and to be granted mutually exclusive access to shared data. It saves time when executing code that rarely has two processors simultaneously requesting the same mutual exclusion lock. The basic idea of optimistic synchronization is to assume the lock is free, send a non-blocking lock request, and continue execution within the mutual exclusion section. Immediate continuation is safe since the group root can suppress propagation of improper data changes that occur if the lock request cannot be granted before data are changed. The root is both the lock owner and the sequencing arbiter for all data changes within the group. Any improper changes to local memory are corrected by rollback.

To prevent unnecessary rollbacks, at a minimum each processor checks that its local memory shows the lock to be free before issuing a new request. However, local lock status may be out of date. A more accurate test is whether on average the local lock copy has shown little indication of recent use. If so, the lock probably is currently unused even in huge networks with long round-trip delays, and a request for optimistic mutual exclusion can be used. If not, the requestor can proceed with a regular lock request. This method does not add any network traffic when the lock is heavily contended.

Using the new algorithm, if there is little indication of recent lock use, the processor assumes a free lock, sends a non-blocking lock request, and optimistically continues execution. However, the compiler must generate code to support execution rollback in case the lock is busy when the request reaches the lock owner. The processor continues computations in the mutual exclusion section while the lock request is propagating. In the best case, lock permission will have arrived before the computation finishes. If so, the processor can then release the lock and continue execution. Useful code execution will have overlapped all communication delays in getting permission. If not, the processor will have finished some needed work before waiting. It can restore the prior values of all changed variables if another processor gets lock permission first.

The code segment in Figure 3 covers a typical update: a shared variable is read, used for local computations, and written back. The statements reading the shared data are the key to correctness of the computation. If each statement reads the same valid value, the computation results will be correct. Because of GWC write ordering, when local lock permission has been received from the lock owner, each local shared value is valid since all changes by other processors must have preceded the permission.

```
Get_lock
lcl_c = shared_a + lcl_b + lcl_c
shared_a = shared_a + lcl_c
Release_lock
```

Figure 3: Mutex Code (Read, Compute, and Write)

If a shared datum value is invalid or changes after it is read, the computation results are not correct and a rollback must occur. Any local or shared variable that will be changed must be saved for possible rollback. Figures 4 and 5 show the mutex code segment in Figure 3 transformed (by a compiler) to support rollback. The lines are numbered for reference in the correctness proof that follows in the next section. The *saved_* prefix demarks variables that must be stored before optimistic calculations.

The outermost *if* (01) in Figure 4 is to prevent a processor from attempting to reacquire a lock that it is already holding. If there is no nested mutual exclusion, a flag is set indicating that variables have not been saved for rollback(02) and a lock request(03,04) is sent. The lock request(04) is an atomic exchange of the value in the local lock copy with a temporary variable that holds the -(processor id) request value. Requesting the lock and saving the previous local lock value access the same memory location and must be atomic lest a new lock value arrive after storing of the old value and before setting of the lock request value. Such a new lock change would be lost. Ignoring it could allow another processor to change shared values during local saving for rollback. If inconsistent values were used for rollback, invalid behavior could result.

The temporary variable(old_val) indicates the local state of the lock. It can also be used to build up a frequency history(05) for a more accurate test of the lock usage. The history frequency information can, as an example, be derived from a simple formula such as $old = 0.95*old + 0.05*new$, where *old* and *new* represent usage and 1.0 means "lock held by another CPU". Frequency history is also updated during an interrupt (P9) where another processor gets the lock.

After issuing a lock request and collecting history information, the processor sets an interrupt so it is warned whenever the lock is changed, either to free the lock before the next lock request arrives or to grant lock permission to this or another processor. The interrupt is atomically coupled with a suspension of insharing for data and lock values. The hardware interface enforces this temporary blocking of external changes to local memory. Without insharing suspension, a shared value immediately following the grant of the lock to another processor may be lost by writing a local memory location just before a rollback overwrites it. However, it is acceptable for the lock to flicker on (grant to another) and off (free) in the instant(01-05) before the interrupt is set, since no rollback values have been saved. Figure 5 shows the interrupt code.

The second *if* statement(07) in Figure 4 checks whether the local lock copy indicates current or recent locking. If the history value (old) is above a certain threshold (e.g. 0.30), meaning the central copy is possibly locked, a processor takes the "regular" path of waiting or swapping context until lock permission has

```
(01) if local_copy not local CPU then {
(02) variables_saved = NO                      /* Variables not yet saved          */
(03) old_val = -id                             /* Will request lock when shared    */
(04) atomic_exchange(old_val,local_copy)       /* Save old value and request lock  */
(05) update usage frequency history
(06) enable intrpt_and_sharing_suspension      /* Watch for another CPU            */
(07) if local_copy or old_val or history indicate usage
(08)   then                                    /* Regular lock request             */
(09)     disable intrpt_and_sharing_suspension
(10)   reg_wait: wait for lock grant           /* Wait or context swap             */
(11)     lcl_c = shared_a_in + lcl_b + lcl_c   /* Regular code execution           */
(12)     shared_a_out = shared_a_in + lcl_c
(13)   else                                    /* Optimistic lock request          */
(14)   opt_calc: saved_lcl_c = lcl_c           /* Save changing variables          */
(15)     saved_shared_a_in = shared_a_in
(16)     variables_saved = YES
(17)     lcl_c = shared_a_in + lcl_b + lcl_c   /* Calculate                        */
(18)     shared_a_out = shared_a_in + lcl_c    /* Lockmngr may stop shared_a       */
(19)     wait until lock answer with local ID  /* Wait or context swap             */
(20) endif
(21) goto finish
(22) roll_back: shared_a_in = saved_shared_a_in       /* Yes, saved, restore vars  */
(23)     lcl_c = saved_lcl_c
(24)     variables_saved = NO                  /* Reset saved variables flag       */
(25)     resume insharing                      /* Back to business                 */
(26)     goto reg_wait
(27) finish: release lock (local_copy = free)  /* Successful execution             */
(28) } else ERROR(Cannot safely nest mutex lock requests)
```

Figure 4: Compiler Generated Code

been granted. The regular path *must* be taken if the local copy shows current lock usage, lest another CPU change shared values while rollback saving occurs.

The case when the lock appears unused is more interesting. The processor continues execution (14) by saving variables that will be changed. In Figure 4, *saved_lcl_c* is the copy of local variable *lcl_c*. After all variables that will be changed are saved, but before any are altered, the *variables_saved* flag is set to *YES* to indicate that a later rollback is feasible.

Shared values like *shared_a* can safely be changed optimistically since all changes must pass through the group root, which is the lock manager. If the local CPU does not have the lock when the new values reach the root, it will discard them. In huge networks, safe preposting of shared changes is usually the major source of benefit from optimistic locking. Possibly long lock grant delays are overlapped with proportionally long sharing propagation times and with any minor increases in mutual exclusion code execution time needed to implement the protocol.

Whenever a lock change arrives (a special location), the interrupt code at *intrpt_and_sharing_suspension* in Figure 5 is activated. If the lock is being freed (momentarily) or being granted to the local processor, mutex execution continues. If another processor now has the lock, values used for calculations may have been incorrect and any changed shared values must be restored. The free indication can follow previous lock use by either another processor (before line 07) or the local one. If local, any shared data changes echoed by the root will be dropped by the sharing interface and not disrupt memory during saving for rollback. Echoed local lock changes are part of the same mutex group as their data, but are not dropped.

The interrupt and suspension of insharing are atomic lest a new shared value immediately follow new lock permission to another processor and be overwritten by the subsequent rollback as it restores prior local values. After the rollback, insharing resumes, any suspended value change packets from the group arrive in memory, and the processor waits (or context swaps) until its lock permission arrives before again calculating values in the mutual exclusion section.

```
(P0) intrpt_and_sharing_suspension:            /* Lock changed; interrupt    */
(P1) disable intrpt_and_sharing_suspension /* and suspend insharing      */
(P2) if lock grants permission for local CPU or shows lock is free
(P3)     then
(P4)         if lock grant is free
(P5)             enable intrpt_and_sharing_suspension
(P6)         resume insharing                   /* Continue insharing         */
(P7)         return where called
(P8)     else
(P9)         update usage frequency history
(PA)         if variables_saved == NO
(PB)             then resume insharing          /* Continue insharing         */
(PC)                 return to reg_wait
(PD)             else                           /* Optimistic failed; rollback */
(PE)                 return to roll_back
```

Figure 5: Interrupt Code

```
(H1) hw_block:
    /* Drop echoed local values for shared variables */
    /* so not overwriting rollback values            */
(H2) if packet from local processor and
(H3)     packet is data in mutex group
(H4)         then drop the packet
```

Figure 6: Hardware Blocking Mechanism

After restoration, without the hardware blocking mechanism in Figure 6, possibly wrong values that resulted from optimistic changes by the local processor may be echoed back from the group root and overwrite newly arrived valid values sent by another processor. The local memory sharing hardware mechanism shown in Figure 6 deletes all incoming locally originated packets targeted to any mutex group, eliminating the problem of bad "overwriting" packets. The sharing interface drops all root-echoed changes to shared local variables written only under a mutual exclusion lock. These echoed values are not needed since only one processor at a time can write these variables and locally stored values are changed in the proper group write order whenever this processor holds the lock. However, ordinary non-mutex values must be echoed to achieve GWC order on all participating processors.

Other processors need not block wrong local packets since any invalid values that pass through the group root will do so after that root has granted lock permission to the local processor. Any other processor either will already have indication of the new lock permission or will be interrupted if it were also trying to perform optimistic mutual exclusion. In either case, because of locking, it will not be able to access any invalid values before they are corrected.

The hardware blocking mechanism in Figure 6 also eliminates the problem when a processor locally releases the lock, but reenters the optimistic mutual exclusion section before the official *free* returns. Any previous local changes to shared values precede the free indicator. Without hardware blocking they might cause inconsistency in rollback saving. For example, if the same variable were written twice in a mutual exclusion section and only the first change had returned before saving, the rollback values would be improper.

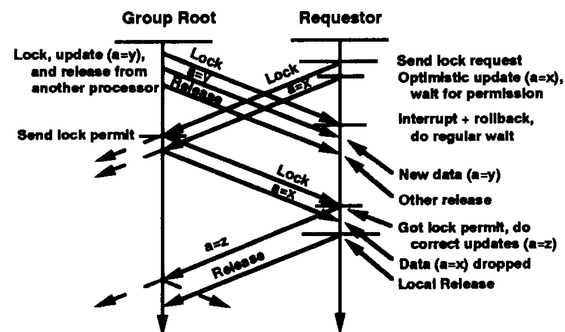

Figure 7: The Most Complex Rollback Interaction

Figure 7 illustrates a problem solved by hardware blocking. After a processor sends a lock request and optimistically updates a variable $a = x$, where x depends on a, it waits for lock confirmation. During the wait and before the local lock request reaches the

group root, another processor's lock request, its update of $a = y$, and its lock release reach the root. The arrival of the other lock grant causes interrupt and rollback on the local processor. All incoming data arrivals are suspended during rollback to prevent the destruction of new valid data.

After rolling back(22), the processor waits(10), periodically checking the lock value until the needed local lock permission arrives. Once it has the lock, the local processor makes the correct updates ($a = z$)(11,12) and releases the lock(27). Hardware blocking will drop any incorrect values ($a = x$). In Figure 7, the time between the correct update and the release is shown to be unusually long, just to make the picture clearer.

4.1 Speedup with Optimistic Locking

A simple example has been constructed to show how optimistic mutual exclusion can shorten execution times. Each processor repeatedly waits for data from processor $i-1$, performs local computations, gets a lock, performs more local computations and updates shared data in a mutually exclusive section. After releasing the lock, it calculates new data and shares it with processor $i+1$. Processor i then continues local calculations before looping again. This example is basically a linear pipeline of events, where two sets of local calculations can overlap at a time. There is no contention among the processors for the mutually exclusive section, so no rollbacks occur. It shows the potential of optimistic synchronization over non-optimistic synchronization in speeding code execution.

The computation time ratio of the mutual exclusion section to each local computation outside it has been selected to be $\frac{1}{8}$. With this ratio, the mutual exclusion section execution time is smaller than the local task time, but not so small that local calculations completely dominate. The time for the mutual exclusion section has also been chosen so communication delay to request the lock for optimistic mutual exclusion can initially be overlapped by calculations.

Figure 8 shows simulation results predicting equivalent network power when running this constructed example on up to 128 processors for data size 1024, so there are from 1024 to 8 iterations of the main loop. Each processor is assumed to have a peak computation speed of 33 MFLOPS and a 400 MB/sec local memory bandwidth. Network power is the product of average sustained efficiency on each processor times the number of processors. It shows how much faster the parallel code runs for each network size.

The top line in Figure 8 shows the maximum network speedup (1.89 for 2 or more processors) possible

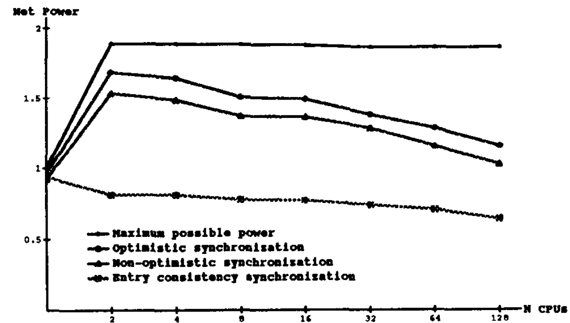

Figure 8: Mutex Methods (Network Power in CPUs)

if there were no network delay in the communication of data and locks between processors. Linear pipelining keeps the maximum below 2. The other lines include the delays for each method assuming that each data sharing hop in a square mesh torus takes $200ns$, and each point to point fiber link is 1 gigabit/sec.

The second line (1.68 for 2 CPUs, 1.15 for 128) shows how optimistic mutual exclusion compares to the already very fast non-optimistic mutual exclusion shown in the third line(1.53 for 2 CPUs, 1.03 for 128). As network size grows, communication delays for locks and data become longer and the mutual exclusion section overlaps less of the lock request delay.

Entry consistency is shown in the bottom line in Figure 8. Entry consistency does not show favorable results (0.81 for 2 CPUs, 0.64 for 128) for two reasons: extra time is needed to transmit the shared data in the mutual exclusion section and demand fetch is needed when non-mutually exclusive data is read.

The peak values of network speedup occur for two processors and are 1.89 with no delays, 1.68 for optimistic locking, 1.53 for non-optimistic GWC locking, and only 0.81 for entry consistency. In summary, this section shows that execution with optimistic synchronization can be 1.1 times faster than with non-optimistic locking under group write consistency and 2.1 times faster than with entry consistency.

5 Conclusions

This paper has introduced optimistic lock synchronization under the group write consistency model as developed for the Sesame sharing network interfaces. GWC guarantees the ordering of writes within a processor group for eagersharing distributed memories. The method uses a local copy of a requested lock to

determine whether the lock is likely to be free. If not free, a regular lock request is made and either a context swap or a busy wait occurs. If free, an optimistic lock request is made and execution continues. If the lock request is not granted, rollback recovers from any improper execution. In particular, optimistically written new values for shared variables will have been stopped by the group root if the source node did not yet have lock permission.

When successful, optimistic lock synchronization hides all or part of the delays needed to obtain lock permission from a remote lock owner. The timings example in Figure 8 illustrates that local release can occur up to 1.5 times faster with non-optimistic mutual exclusion under GWC than with entry consistency. Optimistic synchronization can be up to 2.6 times faster. The code for Figure 8 shows a speedup of 2.1 for optimistic synchronization over entry consistency and of 1.1 over non-optimistic synchronization also under GWC. The combination of optimistic lock synchronization and group write consistency can let a processor execute mutual exclusion code with little or no wasted time in acquiring and releasing the lock.

References

[1] T.E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. *International Conference on Parallel Processing*, II:170–174, August 1989.

[2] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. TR#CMU-CS-91-170, Carnegie Mellon, Sept. 1991.

[3] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, Coherence and Event Ordering in Multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[4] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. *15th Ann. Int. Symp. on Comp. Arch.*, 373–382, May 1988.

[5] D. Lenoski et al. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. *Spring COMPCON 90*, 62–67, February 1990.

[6] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *The 17th Ann. Int. Symp. on Computer Architecture*, 15–26, May 1990.

[7] H. Garcia-Molina and A Spauster. Ordered and Reliable Multicast Communication. *ACM Trans. on Computer Systems*, 9(3):242–271, August 1991.

[8] J.R. Goodman. Cache Consistency and Sequential Consistency. TR#1006, Computer Science, University of Wisconsin, February 1991.

[9] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. *3rd Int. Conf. on ASPLOS*, 3:64–75, April 1989.

[10] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[11] G. Hermannsson and L. Wittie. Fast Locks in Distributed Shared Memory Systems. *27th Ann. Hawaii Int. Conf. on System Sciences*, I:574–583, January 1994.

[12] Sparc International Inc. *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1992.

[13] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[14] J.M. Mellor-Crummey and M.L. Scott. Synchronization Without Contention. *4th Int. Conf. on ASPLOS*, 269–278, April 1991.

[15] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[16] U. Ramachandran, M. Ahamad, M. Yousef, and A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. *International Conference on Parallel Processing*, II:160–169, August 1989.

[17] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. *11th Ann. Int. Symp. on Comp. Architecture*, 340–347, June 1984.

[18] L.D. Wittie, G. Hermannsson, and A. Li. Eager Sharing for Efficient Massive Parallelism. *1992 International Conference on Parallel Processing*, II:251–255, August 1992.