PAPER

Delaying Coherence Requests to Enhance the Performance of Strict Consistency Models

Young Chul SOHN[†], **NaiHoon JEONG**[††], **Jin-Soo KIM**[†], *and* **Seung Ryoul MAENG**[†], *Nonmembers*

SUMMARY Advances in ILP techniques enable strict consistency models to relax memory order through speculative execution of memory operations. However, ordering constraints still hinder the performance because speculatively executed operations cannot be committed out of program order for the possibility of mis-speculation. In this paper, we propose a new technique which allows memory operations to be non-speculatively committed out of order without violating consistency constraints. Consistency constraints are guaranteed through delaying the coherence requests. The proposed technique also improves the performance of spin lock primitives such as TTS lock or MCS lock. Through delaying early acquire requests, the lock transfer time can be improved when there is high contention for a lock.

key words: *multiprocessor, distributed shared memory, memory consistency model, ILP*

1. Introduction

The memory consistency model is a crucial factor for the performance of shared memory multiprocessor systems. Strict consistency models (such as sequential consistency (SC) or total store ordering (TSO)) offer intuitive programming interface, but the inability to perform memory operations out of program order limits the performance.

Modern microprocessors incorporate techniques to exploit instruction-level parallelism (ILP). ILP techniques enable aggressive optimizations [2], [4], [14] for strict consistency models, which relax memory order speculatively. Gharachorloo et al. [2] proposed hardware prefetching and speculative load execution. These two optimizations significantly improved the performance of strict consistency models through issuing memory operations out of program order. However, ordering constraints in strict consistency models still hinder the performance [14]. First, the store-to-load ordering (in SC) prohibits a load from bypassing prior stores and retiring from the reorder buffer. It may cause a high latency store to block the instruction flow through the reorder buffer. Second, the store-to-store ordering (in SC or TSO) forces stores to be performed one after another. It may cause underutilization of memory units and cache ports.

To alleviate above problems, speculative retirement [14] and SC++ [4] are proposed. Those techniques allow memory operations to be speculatively *committed** (retired) out of program order. Thus, memory operations no longer stall processor pipelines waiting for the completion of prior operations. However, out-of-order commitment may incur a consistency violation. To recover from possible consistency violations due to speculative commitment, a processor should store the architectural state into an additional history buffer and roll back when a mis-speculation is detected.

The effectiveness of the speculative commitment techniques is mainly limited by the size of a history buffer; the number of instructions can be committed speculatively at a time. The size of a history buffer should scale with the performance gap between processor and memory subsystem. To hide the higher memory latency, a processor should provide the larger history buffer [4]. Especially, in case of a speculative commitment of a load, the history buffer should keep rollback information not only for the load itself but also for all of the instructions following the load.

In this paper, we propose a new mechanism, the request reorder buffer (RRB) technique, to alleviate the impact of the store-to-load and store-to-store ordering constraints in strict consistency models. The RRB technique enables non-speculative commitment of memory operations (both loads and stores) bypassing prior stores. To guarantee consistency constraints, the RRB technique rearranges the global order of memory operations by delaying a cache coherence request. Because the RRB technique does not require rollback, long memory access latencies can be hidden with small cost of storage, and the negative effects of the speculative techniques can be removed. The RRB technique introduced in this paper can also enhance the performance of lock hand-off for a contended lock. By out-of-order commitment of release operation, we can reduce the negative impact of *early acquires* on a contended lock.

The rest of the paper is organized as follows. Section 2 describes the current optimizations for strict consistency models. Section 3 describes the RRB technique. Section 4 describes how the RRB technique can improve the performance of contended lock. In Sect. 5, we report experimental results of the RRB technique. Section 6 concludes the paper.

Manuscript received March 10, 2003.
Manuscript revised August 11, 2003.

[†]The authors are with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373–1 Guseong-Dong, Yuseong-Gu, Daejon 305–701, Korea.

[††]The author is with NCsoft corp., Seung Kwang Bldg, 143–8 Samsung-dong, Kangnam-gu, Seoul 135–090, Korea.

*We call an operation is *committed* when it updates the processor and memory state; a load is committed when it update destination register and is retired from the reorder buffer, and a store is committed when it updates the cache and is removed from the store buffer (or load/store queue).

2. Background

ILP processors execute multiple independent instructions concurrently and potentially out of program order. To maintain precise interrupts [15], all inflight instructions are stored in the reorder buffer in program order, and modify the architectural state of the processor when they retire from the reorder buffer.

Though ILP technique allows memory operations to be executed out of order as long as they do not access the same location, ordering constraints of memory consistency models prohibits reordering memory operations. Especially in case of strict consistency models such as SC and TSO, the inability to reorder memory operations hinders the performance because long latencies of memory access can not be hidden through overlapping memory operations. On the other hand, relaxed models such as RC [3] allow most of operations to be executed out of order except at synchronization points.

To remedy this problem, advanced techniques such as hardware prefetching, speculative load execution [2], and store buffering [1] were proposed. With hardware prefetching technique, non-binding prefetches for memory operations in the reorder buffer can be issued to hide memory access latency. Moreover, speculative load execution allows the prefetched value to be consumed by a load and subsequent instructions. Store buffering allows pending stores to retire from the head of the reorder buffer. With these techniques, strict consistency models can emulate the behavior of RC and the performance gap between consistency models significantly narrowed. However, there still remains the room for improvement.

Figure 1 shows a typical snapshot of a MIPS R10000 [17]-like ILP processor which incorporates above techniques. Suppose that the program segment in left side is executed and operation l1 misses in the cache while l3 and l4 hit. Because of store buffering, l1, l2 and l3 retire from the reorder buffer although l1 is still pending.

In this situation, two types of ordering constraints may hinder the performance. Firstly, in SC, though l4 is complete, it cannot retire (committed) from the reorder buffer until l1 and l3 are complete because of the store-to-load ordering constraint. Thus, the processor pipeline may stall if the store is not complete before the reorder buffer is filled up. In RC or TSO, however, l4 can be committed as soon as it reaches to the top of the reorder buffer bypassing prior stores.

Secondly, in SC and TSO, though l3 hit in the cache and ready to be issued, it cannot be performed until l1 is complete due to the store-to-store ordering constraint. Because stores should be performed one at a time in program order, they may generate a bursty traffic to the cache if a long latency store at the head of the load/store queue blocks multiple subsequent stores. And it may also cause the load/store queue to fill up, which may also produce pipeline stalls. In RC, l3 can be performed bypassing l1 and removed from the load/store queue.

3. The RRB Architecture

This section describes the RRB technique. We assume a cc-NUMA system similar to SGI Origin2000 [7]. Processors incorporate hardware prefetching, speculative load execution and store buffering. Also, an invalidation-based cache coherence protocol is used. We will describe how the RRB technique works in SC, and it can be directly applied to TSO.

3.1 Delaying Coherence Request with the RRB Technique

Memory consistency models specify the order in which operations *appear* to be executed, not necessarily the order in which they are *actually* executed in a system. Thus, out-of-order execution of operations does not violate consistency constraints as long as the result of the execution is the same as if the operations were done in an order which is consistent with program order.

When an operation x is executed out of order, x appears to be executed in program order if no *conflict operation* is executed until all of the operations prior to x are complete. Two operations *conflict* if they access the same location, and at least one is a store.

In the Fig. 2, assume $y_0, \ldots, y_n$ and x are memory operations in program order. The operation x is executed out of order (denoted as x') bypassing $y_0, \ldots, y_n$ and operations prior to x are complete at t_1. In this case, if an operation z which conflicts with x is executed at t_2, the out-of-order execution x' appears to be executed in program order. It is because the in-order execution of x (denoted as x'') produces the same result as x'. On the other hand, if z is executed at t_0, x' and x'' produce different results because conflict operations produce different results if the execution order is changed – (x, z) and (z, x) produce different results. In

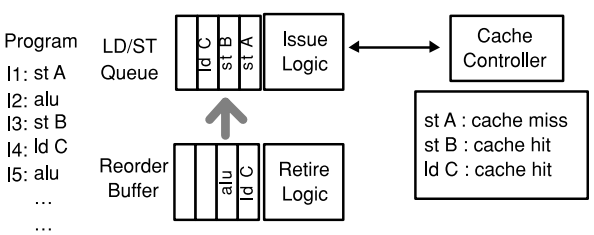

Fig. 1 A snapshot of an ILP processor.

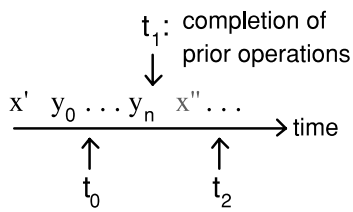

Fig. 2 The correctness of the out-of-order execution.

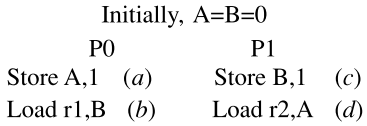

Fig. 3 The sample program.

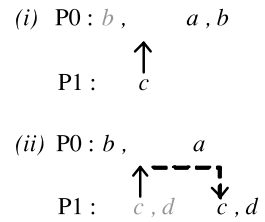

Fig. 4 The case when out-of-order execution may cause a consistency violation. (i) In the previous schemes, processor discards the out-of-order execution and re-issue b. (ii) We can guarantee that c will never be performed between b and a by delaying the coherence request of c.

this case, the out-of-order execution of x may violate consistency constraints.

For example, in the sample program in Fig. 3, suppose that P0 executes b before a. If the operations are executed in the order of (b, a, c, d), the operation b appears to be executed in program order (a, b, c, d) because no conflict operation is executed between b and a.

However, if c is executed between b and a, it is not always possible for b to appear to be executed in program order. For example, if c and d are performed between b and a, (b, c, d, a), the result of the out-of-order execution of b, r1 = 0, is different from the in-order execution of b, (c, d, a, b), which produces r1 = 1. In this case, the result, r1 = 0, should be discarded and b should be re-issued after a because the result of r1 = r2 = 0 violates sequential consistency. To avoid this inconsistency, in the speculative retirement [14] and SC++ [4], P0 nullifies the result of the out-of-order execution of b when it receives the invalidation request of c, then re-issues b (Fig. 4 (i))†.

On the other hand, if the invalidation request of c is delayed until a is complete, we can avoid the re-issue of b. By delaying the invalidation request, we can guarantee that the result of the out-of-order execution of b is the same as that of the in-order execution of b because c will never be executed between b and a–we can avoid an incorrect execution (b, c, d, a) and enforce an execution (b, a, c, d), which is consistent with program order (Fig. 4 (ii)). Note that delaying coherence request does not affect the correctness of the program because operations from different processors can be performed at any order.

As seen by above example, the out-of-order execution of an operation does not violate consistency constraints as long as the coherence request (invalidation, downgrade, and replacement) to the accessed block is delayed until prior operations are complete. Because the operation does not need to be re-issued, the operation can be committed out of program order. In this paper, we propose the RRB technique, which allows an operation to be committed out of order.

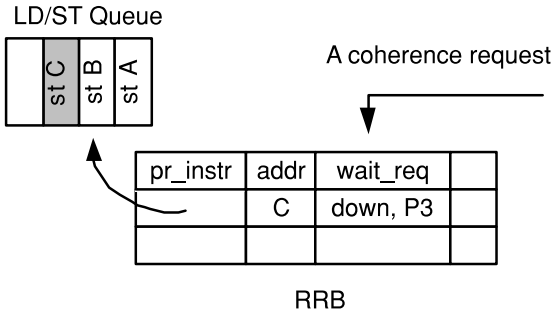

Fig. 5 Example of out-of-order commitment.

Unlike the previous schemes [4], [14] which rely on speculative commitment and rollback, proposed scheme *non-speculatively* commits an operation while consistency constraints are guaranteed by delaying coherence requests.

When a completed load reaches the head of the reorder buffer before prior stores are complete, the load is committed (retired) bypassing stores. Similarly, when a store is ready to be issued but should wait for the completion of prior stores, the store is performed (committed) to the cache bypassing stores. Whenever a load or a store is committed bypassing prior stores, processor should guarantee that no other processor executes a conflict access until prior stores are complete.

The request reorder buffer (RRB), which is a special buffer between processor and cache controller, takes charge of delaying coherence requests. When a processor commits an operation out of order, the address of the accessed cache block is stored in the RRB. Whenever a coherence request is received, cache controller should check the RRB before it processes the coherence request. If the target address of the coherence request matches in the RRB, the request is delayed until the prior operations of the committed operation (pointed by a field in the RRB) are complete. Figure 5 shows an example of out-of-order commitment using the RRB technique. If store C is committed out of order, the address C is registered in the RRB. When a coherence request to C is received, the request is removed from cache pipeline and stored in the RRB until prior stores are complete. Figure 5 depicts a situation that P3 executes load C and the downgrade request of load C is delayed by the RRB technique.

Note that on a replacement of cache block which is registered in the RRB, the replacement request should also be stored in the RRB. It is because once a block is replaced from a cache, a coherence request is no longer delivered to the processor, which may result in a consistency violation.

†Note that (b, c, a, d) does not violate sequential consistency though c were executed between b and a. It is because there exists an execution order (a, b, c, d) which produce the same result as (b, c, a, d). However, it is hard to detect whether d is executed before a or not. Thus, the previous schemes conservatively assume that the execution of c as the consistency violation.

3.2 Deadlock Avoidance

Because an operation may be delayed indefinitely, the RRB technique may cause a deadlock. In the sample program in Fig. 3, suppose that P0 commits b bypassing a and the invalidation request of c is delayed at P0. There is a wait-for dependency between a and c, which is denoted by $a \rightarrow c$: c waits for the completion of a. In this situation, if P1 also commits d out of order, the invalidation request of a should be delayed until c is complete. Thus, there is also a wait-for dependency $c \rightarrow a$. This cyclic dependency leads to a deadlock situation. In this paper, we propose a deadlock avoidance scheme which limits the out-of-order commitment based on the address of the memory block. From now on, we will denote the address of memory block accessed by an operation x by '$\&x$'.

Deadlock avoidance scheme: Processors are allowed to commit an operation x bypassing an operation y, if and only if $\&x > \&y$.

Correctness: To perform the proof by a contradiction, suppose that the RRB technique with proposed deadlock avoidance scheme makes cyclic wait-for dependencies among several processors as follows.

$$a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow a_0 \quad (a_k\text{: operations}) - \qquad (1)$$

A wait-for dependency $a_k \rightarrow a_{k+1}$ implies that there exists an operation x which access the same memory location with a_{k+1} (i.e. $\&x = \&a_{k+1}$), and x is committed bypassing a_k. By proposed scheme, x can bypass a_k if and only if $\&a_k < \&x$. Thus, $\&a_k < \&a_{k+1}$. Generally, the supposed dependency (1) means

$$\&a_0 < \&a_1 < \ldots < \&a_n < \&a_0$$

It is a contradiction. Therefore, there is no cycle caused by out-of-order commitment. ■

As an example, in Fig. 3, if $\&b > \&a$, P0 is allowed to commit b bypassing a but P1 cannot commit d out of program order. Thus, cyclic wait-for dependency is not created and deadlock is avoided.

Although the proposed scheme avoids deadlock, it limits the performance because some of the operations may not be committed out of order due to the deadlock avoidance condition. In general, the more operations are committed out of order, the higher performance is achieved. Because the proposed scheme restricts out of order commitment based on the target address of an operation, the performance is highly dependent on the memory access patterns of an application. The proposed scheme performs well on applications of which the addresses of long latency stores are usually lower than addresses of following operations.

We expect that a more sophisticated deadlock avoidance scheme could achieve the more performance enhancement through, for example, exploiting memory access patterns of applications or relocating addresses of memory blocks by compilers.

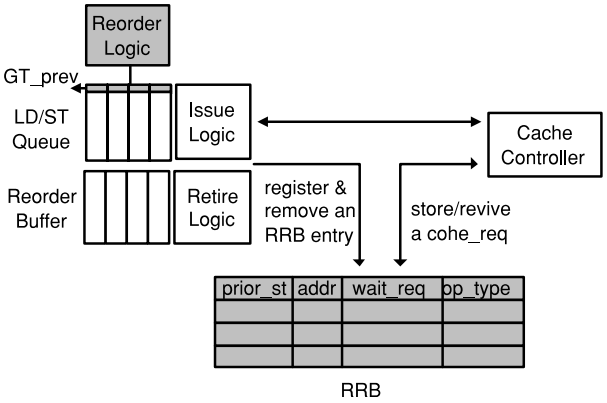

Fig. 6 The block diagram of the RRB architecture.

3.3 Implementation of the RRB Technique

Figure 6 shows the block diagram of processor architecture for the RRB technique. For simplicity, we present logic blocks related to the RRB technique but omit other components. The load/store queue is similar to the address queue of MIPS R10000 processor [17], which holds memory operations in program order. The retire logic commits the result of completed operations in the reorder buffer. The issue logic actually performs memory operations in load/store queue. Without the RRB technique, the retire logic cannot retire a load bypassing stores and the issue logic should perform stores one after another in the program order.

In the RRB technique, loads and stores are committed bypassing prior stores. To delay the coherence request which conflicts to the committed operation, an RRB entry for the committed operation is created.

An RRB entry consists of four fields; *addr* field which stores the block address of a committed instruction, *prior_st* field which points to the nearest prior store to the committed operation in the load/store queue, *wait_req* field which stores the delayed coherence requests. Maximum two coherence requests can be stored in *wait_req* field; one is a coherence request from the directory and the other is a replacement request. Note that once a coherence request is delayed by the RRB, the directory does not send another coherence request to that block until the delayed coherence request is replied. The last field of the RRB is *op_type* field which indicates the type (load or store) of the committed operation. The *op_type* field is required to prohibit useless delays; when a load is committed out-of-order and the accessed block happens to be in the cache with exclusive state, an invalidation request to the block should be delayed, but a downgrade request can be serviced because loads can be executed concurrently. A downgrade request is delayed only if there is an RRB entry whose *op_type* is 'store'.

To implement deadlock avoidance scheme, we add the reorder logic and one-bit *GT_prev* field to the load/store queue; *GT_prev* field indicates whether the operation can be committed out of order with respect to the deadlock avoidance condition. The reorder logic sets *GT_prev* field of each

operation in the load/store queue through comparing the target address of the operation with that of prior operations. If an operation accesses the highest address than all of prior instructions, *GT_prev* field is set indicating the operation can be committed.

Out-of-order commitment of loads and stores are actually done by the retire logic (loads) and the issue logic (stores). If a load reaches to the top of the reorder buffer, the retire logic checks the load/store queue status. If the *GT_prev* field of the load is set, the load is committed even if there are prior incomplete stores. Stores are committed by the issue logic. If there is a store whose *GT_prev* field is set, the issue logic performs the store to the cache. Whenever an operation is committed out of order, the operation is registered in the RRB and removed from the load/store queue.

Whenever a coherence request is received, the cache controller checks the RRB before it processes the coherence request. If the target address and the *op_type* of the coherence request matches with an RRB entry, the request is removed from the cache controller and stored in the *wait_req* field of the RRB entry.

An RRB entry is removed from the RRB when all of operations prior to the operation which is pointed by the *prior_st* field are complete. Whenever an operation is complete and removed from the head of load/store queue, the issue logic should search the *prior_st* field of the RRB. If it matches, the RRB entries are freed and the delayed coherence requests are revived, if any.

For a multiprocessor system with 256 nodes with a 64-bit processor, an RRB entry requires 74 bits as depicted in Table 1. To build the RRB table with 64 entries, 592 bytes of storage is requred. In case of history buffer with 64 entries, it requires about 1152 bytes of storage and similar amount of storage is required to add 64 enties to the reorder buffer [14].

Note that the *wait_req* field is not enough to store the whole replacement request for a modified cache block. When a modified cache block is replaced, the replacement request contains the cache block for write-back to the main memory. In this paper, we assume that the replaced cache block is temporally stored in a internal buffer which is used for handling replacement requests. In the RSIM abstraction [11], they assume the WRB (write-back buffer) structure to correctly handle possible replacement incurred by incoming reply. The WRB keeps the replaced cache block until the replacement request can be successfully injected into the memory subsystem. Thus, we can utilize the already existing buffer space for storing the replaced cache block.

Table 1 The RRB fields.

Field	Bits	Description
prior_st	6	indicates one of entry in the load/store queue.
addr	59	cache block address which is accessed by an operation which is committed out of order. We assumed 32 byte cache line.
wait_req	12	coherence request which is delayed by the RRB technique. For a external coherence request, 2 bits of type and 8 bits of the requestor's node ID is required. For a replacement request 2 bits of request type is required.
op_type	1	indicates the operation type of committed operation.

When the RRB technique is implemented in cc-NUMA systems, it would complicate the design of the memory subsystem because resource contention in the memory subsystem may incur a deadlock situation. For example, suppose that an operation y is delayed by the RRB mechanism and waits for the completion of x ($x \rightarrow y$). If y occupies a resource which is required for the completion of x, deadlock could occur due to resource contention because x also waits for y to release the resource ($y \rightarrow x$).

Because a detailed implementation of memory subsystem is significantly different for each multiprocessor system, we will describe some of the general techniques to avoid deadlock due to resource contention. One simple method is to release all of resources occupied by a delayed memory operation. It can be implemented by retrying the delayed operation, rather than blocking it at a cache. Because the delayed operation would not hold any resources but release and re-occupy them, a fair arbitration mechanism for resources would guarantee deadlock freedom. Another method is to guarantee the completion of the oldest memory operation of each processor by limiting the number of outstanding operations from one processor. In this method, processors are allowed to issue maximum n operations at a time and the memory system supplies plenty of resources enough for all of the outstanding operations to be performed without contention. If each processor reserves the issue slot for the oldest operation, the oldest operation in a processor can be always issued and never blocked for resources. Thus, we can avoid deadlock.

4. Fast Lock Transfer with the RRB Technique

Efficient locking synchronization primitives are essential for achieving high performance in parallel programs. One function of locking primitives is to enable exclusive access to shared data and critical sections of code.

While many papers suggest various locking mechanisms [6], [9], [13], the most simple and popular mechanisms are spinlocks based on atomic read-modify-write operations (such as TTS (test&test&set) or MCS lock [9]). In those spinlocks, a lock is transfered from a lock holder to a next lock requestor by a store to a lock variable which a lock requestor is spinning on.

For example, in case of TTS lock in Fig. 7, a lock requestor spins on a lock variable until the lock is released. The lock holder release it by clearing the lock variable. If there is no contention for the lock, release operation can be performed within the lock holder's cache. However if other processors try to acquire the lock before it is released, the exclusive ownership for the lock variable is relinquished

from the lock holder. In this case, the lock transfer time would be significantly increased because it requires several roundtrip messages between the lock holder and the requestor as in Fig. 8. Figure 9 (a) shows a best case that the acquire of the next requestor arrives *on time* – right after the release is complete.

As shown in Fig. 8 and Fig. 9 (a), *early acquires* degrade the performance because it increase the lock transfer time and messages in the interconnection network. Thus, if we enforce that the *early acquires* do not invalidate the lock holders cache, the lock transfer time can be significantly reduced. As in Fig. 8 (b), if we delay the coherence request of the *early acquire* until the critical section is end, we can pretend that the acquire has arrived right on time and the lock would be transfered with one-way message.

Rajwar et. al. proposed delaying coherence requests for this purpose in LL/SC- based locking primitives [13].

```
typedef volatile bool TTS_lock;

acquire ( TTS_lock *L )
{
  while ( test&set(L) == LOCKED )
        while ( L == LOCKED ) ;
}

release ( TTS_lock *L )
{
  *L = 0;
}
```

Fig. 7 TTS primitives.

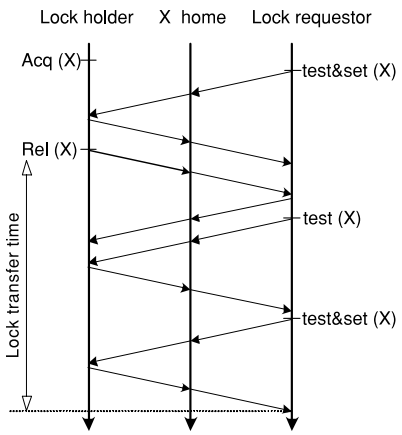

Fig. 8 The negative impact of contention in TTS lock.

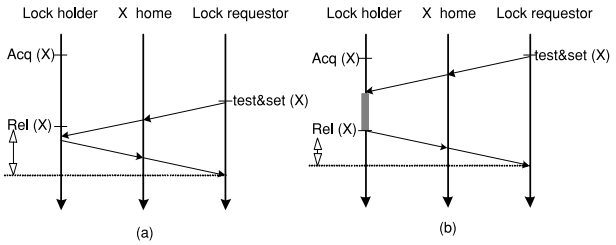

Fig. 9 Reducing lock transfer time. (a) If the acquire arrives on time, the lock can be transfered with one-way message. (b) If the early acquire is delayed, we can pretend the acquire arrived on time.

They modified cache coherence protocol and introduced a new type of cache coherence request (LPRFO) which can be delayed by other processor which is considered as a lock holder. To guarantee forward progress, they rely on time-out mechanism.

In this section, we show that the lock transfer time can be reduced with the RRB technique. As shown in the Fig. 9 (b), we delay the coherence requests of early acquires until the release of the lock holder is complete. Delaying the coherence requests can be implemented without additional modification to the RRB technique, but relocate the address of the lock variable so that the release operation can be committed out of order – thus, the coherence request of the early acquire can be delayed.

Proposed method is effective for a critical section which executes long latency stores in it. Note that proposed method do not require modification to the cache coherence protocol or discrimination of synchronization operation from regular memory operations. Moreover, the method presented in this section can be applied to any spin locks based on atomic read-modify-write operations (such as ticket lock or MCS lock).

4.1 Delaying the Early Acquires

In parallel applications, locks are often used to guarantee threads have exclusive access to shared data for a critical section. Typically, shared location is loaded at the beginning of the critical section, and the location is modified at the end of the critical section. Memory accesses in the critical section usually invoke a cache miss which requires cache coherence request because the locations are actively shared with other processors. Thus, it is common that a critical section is complete when a store at the end of the critical section is complete. Figure 10 shows a typical example of critical section in Raytrace benchmark. In this example, the last operation in the critical section is a store to a shared location and it usually invoke a cache miss which requires remote memory access latency.

In this case, if we commit the release operation as soon as it can be committed (i.e. as soon as it is retired from the reorder buffer), we can remove the negative impact of *early acquires* through delaying the coherence request with the RRB. Note that the release is a store opertion to a lock variable. Figure 11 shows that example of out-of-order commitment of the release in the critical section in the figure. The

```
acquire(global->wplock);

  wpentry = global->workpool[0];
  if (!wpentry)
  {
      global->wpstat[0] = WPS_EMPTY;
      release(global->wplock);
      return (WPS_EMPTY);
  }
  global->workpool[0] = wpentry->next;

release(global->wplock);
```

Fig. 10 An example critical section of Raytrace.

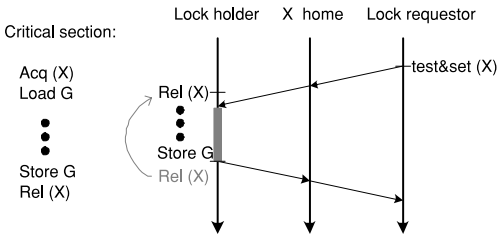

Fig. 11 Out-of-order commitment of a release bypassing prior store.

release is committed bypassing incomplete `store G` which is prior to the release in program order and the coherence request of early acquire is delayed.

To do it, all we have to do is just allocate the lock variable to a higher address range than the address of the location accessed in the critical section (*G* in the Fig. 11), so that the release operation can be committed out of order. It can be done by the application programmer or automatically by the compiler.

Though above scheme improve the performance of lock hand-off, there is a window of vulunerability – if the early acquire is received before the release is committed out of order, the exclusive ownership would be relinquished. Moreover, proposed method does not effective for a critical section which does not have long latency stores in it because we rely on out-of-order commitment of the release bypassing long latency stores.

In the following section, we evaluate the effectiveness of the RRB technique on reducing the lock transfer time.

5. Performance Evaluation

In this section, we evaluate the performance of the RRB technique. We present the evaluation methodology and the results. The performance is measured in terms of overall execution time of selected benchmark applications and microbenchmarks.

5.1 Experimental Methodology

We used RSIM [10] to simulate cc-NUMA system with 16 processors. Table 2 shows the base system configuration. We assume a relatively large L2 cache to eliminate capacity and conflict misses, so that performance difference among the memory models is solely due to the intrinsic behavior of the models. In our experiments, all implementations use non-blocking caches, hardware prefetching, speculative load execution and store buffering. We set the RRB entry size to 64.

Benchmark applications are from the SPLASH2 [16] suite, except for Mp3d from SPLASH [5]. Table 3 gives the input sizes used for the benchmark applications.

To show the effect of fast lock transfer with the RRB technique, we performed an experiment similar to the method used by Kagi [6] and Lim [8]. We constructed a microbenchmark that accesses a critical section in a loop repeatedly (the benchmark accesses the critical section a total

Table 2 Simulated architecture.

System Parameters	
CPU	4-issue per cycle
Reorder buffer	64 instructions
Memory queue	64 instructions
L1 cache	16 KB, direct-mapped
L2 cache	4 MB, 4-way assoc.
Cache line size	32 bytes
L2 fill latency local	41 processor cycles
L2 fill latency remote	117 processor cycles
RRB entry	64 entries

Table 3 Application parameters.

Application	Input Parameters
Radix	512 K keys
Ocean	128x128 ocean
Barnes	4 K particles
Mp3d	50000 particles
Raytrace	Balls4

```
while (loop < 2048/node_num) {

        acquire(global->lock);
        global->sum = global->sum + 1;
        release(global->lock);

        delay (4000 cycles);
        loop++;
}
```

Fig. 12 Microbenchmark used for locking primitives.

of 2048 times which is distributed evenly among the processors). In the critical section a processor execute a load and store to a shared location as in Fig. 12. After release, the releasing processor waits for 4000 cycles.

We simulated four implementations; sequential consistency (SC), total store order (TSO), and adding the RRB technique to SC and TSO (SC+RRB and TSO+RRB, respectively). Note that the performance of TSO is the upper limit of relaxing store-to-load constraint in SC. For the base case, we simulated release consistency (RC). In RC, we do not use the hardware prefetching and speculative load execution because previous results show it is not always beneficial for RC [12].

5.2 Performance of Locking Primitives with the RRB Technique

Figure 13 and Fig. 14 show the total execution time (in cycle) of the microbenchmark of Fig. 12. We measured the performance of TTS and MCS lock with and without the RRB technique. For the +RRB versions, we locate the address of lock variable in the higher address than `global->sum`.

As the number of nodes is increased, the execution time is reduced to some extent by removing the delay of 4000 cycles in the critical path. However, as the contention for the lock increases, and eventually the reduction in execution time is stopped by the increase in the lock transfer time.

As shown in Fig. 13, the RRB technique improves 17.8% of TTS lock and 24.5% of MCS lock in case of 16

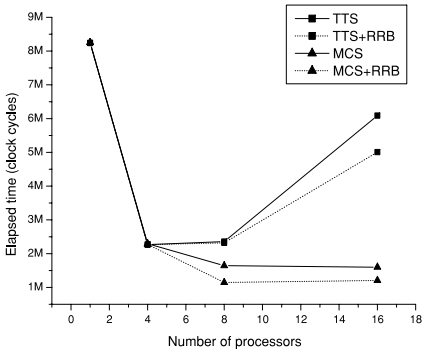

Fig. 13 The effect of fast lock transfer in SC.

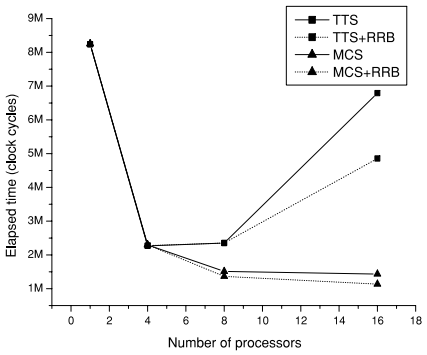

Fig. 14 The effect of fast lock transfer in TSO.

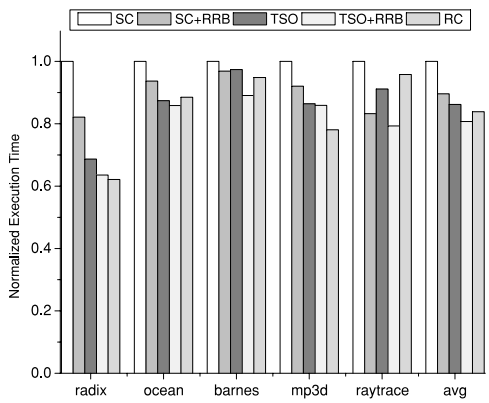

Fig. 15 Normalized execution time relative to SC.

nodes contention.

5.3 Performance Benefits of the RRB Technique

In this section we show the result of the RRB technique in benchmark applications. Throughout the result in this section, the effect of fast lock transfer is included.

Figure 15 shows the execution time of benchmarks normalized to SC. In Radix and Raytrace, 17.9% and 16.8% of the execution time were reduced in SC. In TSO, 7.5% and 13% of the execution time were reduced in Radix and Raytrace. On the average, the RRB technique achieves the performance improvement of 10.5% in SC and 6.3% in TSO.

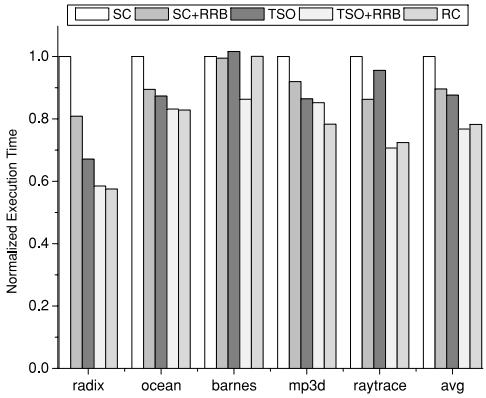

Fig. 16 Impact of network latency.

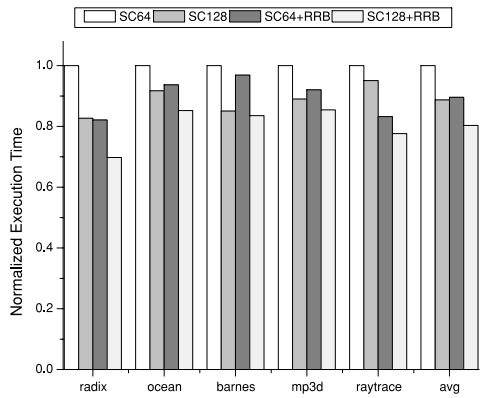

Fig. 17 Impact of reorder buffer size in SC. Each of the implementation labeled with a number indicating the size of reorder buffer.

The performance gap between SC+RRB and TSO is within 3.8%. In Barnes and Raytrace, SC+RRB outperforms TSO because these applications are more sensitive to store-to-store ordering. Ocean, Barnes and Raytrach use locks to synchronize between processors. In these applications, the fast lock transfer with the RRB technique enhance the performance to the extent that TSO+RRB outperforms RC. Comparing to RC, the performance gap between SC+RRB and RC is within 6.4% and TSO+RRB outperforms RC with 3.1%, on the average.

Throughout the simulation, 64 entries of the RRB were sufficient for all cases. In most applications, about half of operations couldn't be committed out of order solely due to the consistency constraints. It is because the proposed deadlock avoidance scheme was too restrictive and sensitive to memory access patterns of applications.

Figure 16 shows the normalized execution time when we increased the network latency to 5 times the remote latency described in Table 2. On increasing the network latency, the performance gain by the RRB technique increased. In TSO, 12.1% of execution time was reduced by the RRB technique on the average. It is due to the fact that as the network latency increases, the processors are more likely to be stalled by long latencies of stores.

Figure 17 and Fig. 18 shows the comparison with in-

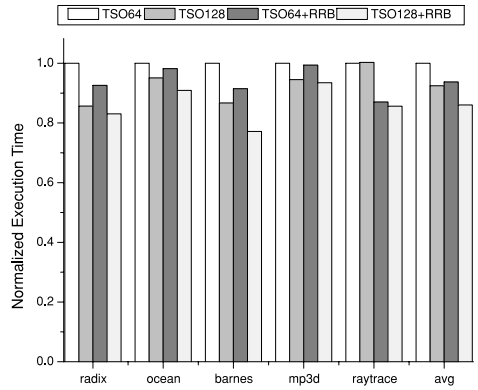

Fig. 18 Impact of reorder buffer size in TSO. Note that the execution time is normalized to TSO64.

creasing the size of reorder buffer. We doubled the size of the reorder buffer (128 entries). We also increased the size of load/store queue to 128 entries. In general, as the size of the reorder buffer is increased, the negative effect of ordering constraints is mitigated. It is because the reorder buffer and the load/store queue are less likely to fill up and hardware prefetches from the reorder buffer can be issued much earlier which effectively reduces memory access latencies. In Fig. 17, SC128 outperforms SC64+RRB in Ocean, Barnes and Mp3d and in Radix and Raytrace, SC64+RRB outperforms SC128. On the average, the RRB technique achieves the similar effect of doubling the reorder buffer in our base system. Figure 18 shows TSO cases. Note that the execution time is normalized to TSO64. In TSO, increasing the reorder buffer is more effective than adding the RRB technique except for Raytrace.

6. Conclusion

We have presented the RRB technique to enhance performance of strict consistency models. To enhance the performance of strict consistency models, previous schemes relax memory order through speculative execution of operations. However, ordering constraints still hinder performance because speculatively performed operations cannot be committed out of order and uncommitted operations frequently exhaust implementation resources.

With the RRB technique, memory operations can be committed out of program order without violating consistency constraints. Moreover the RRB technique effectively alleviate the lock hand-off problem when there is contention for a lock through delaying coherence requests. With a 64-entry RRB, proposed technique achieves maximum 17.9% (SC) 13% (TSO) of performance improvement on our base architecture. The RRB technique reduced the execution time gap between SC and TSO to within 3.8% on the average. Because the RRB technique does not rely on rollback mechanism, storage overhead is small; 592 bytes for a 64-entry RRB.

Current proposal limits the performance gain to avoid deadlock. We expect that deadlock avoidance condition could be more relaxed through exploiting memory access patterns of applications, or relocating address of memory operations by compilers.

Acknowledgment

This research is supported by KISTEP under the National Research Laboratory program.

References

[1] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared memory multiprocessors," Proc. 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), vol.26, pp.245–259, 1991.

[2] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," Proc. 1991 International Conference on Parallel Processing, vol.I, Architecture, pp.I–355–I–364, Aug. 1991.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P.B. Gibbons, A. Gupta, and J.L. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," 25 Years ISCA: Retrospectives and Reprints, pp.376–387, 1998.

[4] K. Gniady, B. Falsafi, and T. Vijaykumar, "Is SC+ILP = RC?," Proc. 26th Annual Int'l Symp. on Computer Architecture (ISCA '99), pp.162–171, 1999.

[5] W.D. Weber, J.P. Singh, and A. Gupta, "Splash: Stanford parallel applications for shared-memory," Computer Architecture News, pp.5–44, March 1992.

[6] A. Kagi, D. Burger, and J.R. Goodman, "Efficient synchronization: Let them eat QOLB," ISCA, pp.170–180, 1997.

[7] J. Laudon and D. Lenoski, "The SGI origin: A ccNUMA highly scalable server," Proc. 24th Annual International Symposium on Computer Architecture (ISCA-97), pp.241–251, June 1997.

[8] B. Lim and A. Agarwal, "Reactive synchronization algorithms for multiprocessors," Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp.25–35, 1994.

[9] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Trans. Computer Systems, vol.9, no.1, pp.21–65, Feb. 1991.

[10] V. Pai, "Rsim: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors," Proc. 3rd Workshop on Computer Architecture Education, 1997.

[11] V.S. Pai, P. Ranganathan, and S.V. Adve, "Rsim reference manual version 1.0," Technical Report 9705, Dept. of Electrical and computer engineering, Rice univ., Aug. 1997.

[12] V.S. Pai, P. Ranganathan, S.V. Adve, and T. Harton, "An evaluation of memory consistency models for shared-memory systems with ILP processors," Proc. 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII), pp.12–23, Oct. 1996.

[13] R. Rajwar, A. Kagi, and J.R. Goodman, "Improving the throughput of synchronization by insertion of delays," HPCA, pp.168–180, 2000.

[14] P. Ranganathan, V.S. Pai, and S.V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," ACM Symposium on Parallel Algorithms and Architectures, pp.199–210, 1997.

[15] J.E. Smith and A.R. Plezkun, "Implementing precise interrupts in pipelined processors," IEEE Trans. Comput., vol.C-37, no.5, pp.562–573, May 1988.

[16] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological consid-

erations," Proc. 22th International Symposium on Computer Architecture, pp.24–36, 1995.
[17] K.C. Yeager, "The MIPS R10000 superscalar microprocessor," IEEE Micro, vol.16, no.2, pp.28–40, April 1996.

Young Chul Sohn received the B.S. and M.S. degree in computer science from the Korea Advanced Institute of Science and Technology in 1994 and 1996, respectively. He is currently towards the Ph.D. degree in computer science at the Korea Advanced Institute of Science and Technology. His research interests include parallel computer architectures and cluster computing.

Nai-Hoon Jeong received the B.S. in computer science from the Yonsei University in 1989. received the M.S and Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology in 1991 and 2002, respectively. He is currently working at NCsoft corporation. His research interests include parallel computer architectures and processor architecture.

Jin-Soo Kim is currently an assistant professor of the department of electrical engineering and computer science at Korea Advanced Institute of Science and Technology (KAIST). Before joining KAIST, he was a senior member of research staff at Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002. He was with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. He received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University in 1991, 1993, and 1999, respectively. His research interests include cluster computing, computer architecture, and operating systems.

Seung Ryoul Maeng received the degree of B.S. in electronics engineering from the Seoul National University, Seoul, Korea, in 1977, and the M.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology in 1979 and 1984, respectively. Since 1984 he has been a faculty member of the Department of electronic engineering and computer science of the Korea Advanced Institute of Science and Technology. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include parallel computer architectures, cluster and grid computing.