

Two Techniques to Enhance the Performance of Memory Consistency Models

Kouros Gharachorloo, Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

The memory consistency model supported by a multiprocessor directly affects its performance. Thus, several attempts have been made to relax the consistency models to allow for more buffering and pipelining of memory accesses. Unfortunately, the potential increase in performance afforded by relaxing the consistency model is accompanied by a more complex programming model. This paper introduces two general implementation techniques that provide higher performance for all the models. The first technique involves *prefetching* values for accesses that are delayed due to consistency model constraints. The second technique employs *speculative execution* to allow the processor to proceed even though the consistency model requires the memory accesses to be delayed. When combined, the above techniques alleviate the limitations imposed by a consistency model on buffering and pipelining of memory accesses, thus significantly reducing the impact of the memory consistency model on performance.

1 Introduction

Buffering and pipelining are attractive techniques for hiding the latency of memory accesses in large scale shared-memory multiprocessors. However, the unconstrained use of these techniques can result in an intractable programming model for the machine. Consistency models provide more tractable programming models by introducing various restrictions on the amount of buffering and pipelining allowed.

Several memory consistency models have been proposed in the literature. The strictest model is *sequential consistency* (SC) [15], which requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Sequential consistency imposes severe restrictions on buffering and pipelining of memory accesses. One of the least strict models is *release consistency* (RC) [8], which allows significant overlap of memory accesses given synchronization accesses are identified and classified into acquires and releases. Other relaxed models that have been discussed in the literature are *processor consistency* (PC) [8, 9], *weak consistency* (WC) [4, 5], and *data-race-free-0* (DRF0) [2]. These models fall between sequential and release consistency models in terms of strictness.

The constraints imposed by a consistency model can be satisfied by placing restrictions on the order of completion for memory accesses from each process. To guarantee sequential consis-

tency, for example, it is sufficient to delay each access until the previous access completes. The more relaxed models require fewer such delay constraints. However, the presence of delay constraints still limits performance.

This paper presents two novel implementation techniques that alleviate the limitations imposed on performance through such delay constraints. The first technique is *hardware-controlled non-binding prefetch*. It provides pipelining for large latency accesses that would otherwise be delayed due to consistency constraints. The second technique is *speculative execution for load accesses*. This allows the processor to proceed with load accesses that would otherwise be delayed due to earlier pending accesses. The combination of these two techniques provides significant opportunity to buffer and pipeline accesses regardless of the consistency model supported. Consequently, the performance of all consistency models, even the most relaxed models, is enhanced. More importantly, the performance of different consistency models is equalized, thus reducing the impact of the consistency model on performance. This latter result is noteworthy in light of the fact that relaxed models are accompanied by a more complex programming model.

The next section provides background information on the various consistency models. Section 3 discusses the prefetch scheme in detail. The speculative execution technique is described in Section 4. A general discussion of the proposed techniques and the related work is given in Sections 5 and 6. Finally, we conclude in Section 7.

2 Background on Consistency Models

A consistency model imposes restrictions on the order of shared memory accesses initiated by each process. The strictest model, originally proposed by Lamport [15], is sequential consistency (SC). Sequential consistency requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Processor consistency (PC) was proposed by Goodman [9] to relax some of the restrictions imposed by sequential consistency. Processor consistency requires that writes issued from a processor may not be observed in any order other than that in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. Sufficient constraints to satisfy processor consistency are specified formally in [8].

A more relaxed consistency model can be derived by relating memory request ordering to synchronization points in the program. The weak consistency model (WC) proposed by Dubois

et al. [4, 5] is based on the above idea and guarantees a consistent view of memory only at synchronization points. As an example, consider a process updating a data structure within a critical section. Under SC, every access within the critical section is delayed until the previous access completes. But such delays are unnecessary if the programmer has already made sure that no other process can rely on the data structure to be consistent until the critical section is exited. Weak consistency exploits this by allowing accesses within the critical section to be pipelined. Correctness is achieved by guaranteeing that all previous accesses are performed before entering or exiting each critical section.

Release consistency (RC) [8] is an extension of weak consistency that exploits further information about synchronization by classifying them into acquire and release accesses. An *acquire* synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations. A *release* synchronization access (e.g., an unlock operation or a process setting a flag) grants this permission. An acquire is accomplished by reading a shared location until an appropriate value is read. Thus, an acquire is always associated with a read synchronization access (see [8] for discussion of read-modify-write accesses). Similarly, a release is always associated with a write synchronization access. In contrast to WC, RC does not require accesses following a release to be delayed for the release to complete; the purpose of the release is to signal that previous accesses are complete, and it does not have anything to say about the ordering of the accesses following it. Similarly, RC does not require an acquire to be delayed for its previous accesses. The data-race-free-0 (DRF0) [2] model as proposed by Adve and Hill is similar to release consistency, although it does not distinguish between acquire and release accesses. We do not discuss this model any further, however, because of its similarity to RC.

The ordering restrictions imposed by a consistency model can be presented in terms of when an access is allowed to perform. A read is considered *performed* when the return value is bound and can not be modified by other write operations. Similarly, a write is considered *performed* when the value written by the write operation is visible to all processors. For simplicity, we assume a write is made visible to all other processors at the same time. The techniques described in the paper can be easily extended to allow for the case when a write is made visible to other processors at different times. The notion of being performed and having completed will be used interchangeably in the rest of the paper.

Figure 1 shows the restrictions imposed by each of the consistency models on memory accesses from the same process.¹ As shown, sequential consistency can be guaranteed by requiring shared accesses to perform in program order. Processor consistency allows more flexibility over SC by allowing read operations to bypass previous write operations. Weak consistency and release consistency differ from SC and PC in that they exploit information about synchronization accesses. Both WC and RC allow accesses between two synchronization operations to be pipelined, as shown in Figure 1. The numbers on the blocks denote the order in which the accesses occur in program order. The figure shows that RC provides further flexibility by exploiting information about the type of synchronization.

The delay arcs shown in Figure 1 need to be observed for correctness. The conventional way to achieve this is by actually delaying each access until a required (possibly empty) set of

¹The weak consistency and release consistency models shown are the WCsc and RCpc models, respectively, in the terminology presented in [8].

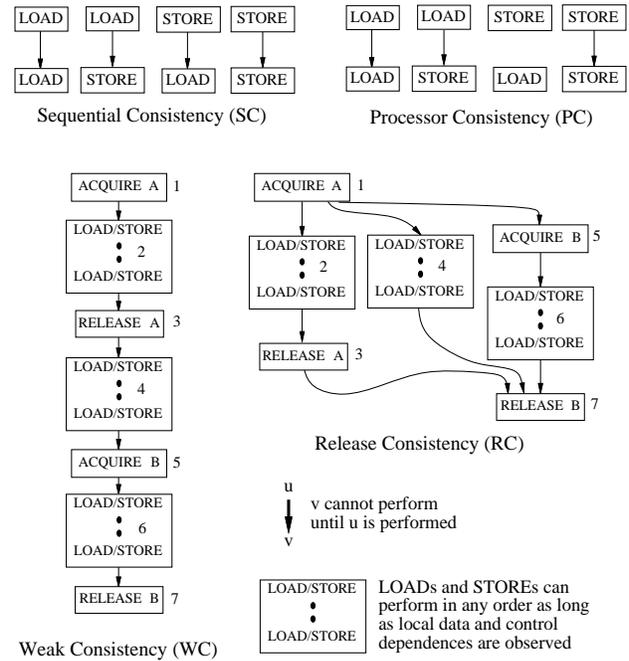


Figure 1: Ordering restrictions on memory accesses.

accesses have performed. An interesting alternative is to allow the access to partially or fully proceed even though the delay arcs demand that the access be delayed and to simply detect and remedy the cases in which the early access would result in incorrect behavior. The key observation is that for the majority of accesses, the execution would be correct (according to the consistency model) even if the delay arcs are not enforced. Such accesses are allowed to proceed without any delay. The few accesses that require the delay for correctness can be properly handled by detecting the problem and correcting it through reissuing the access to the memory system. Thus, the common case is handled with maximum speed while still preserving correctness.

The prefetch and speculative execution techniques described in the following two sections are based on the above observations. For brevity, we will present the techniques only in the context of SC and RC since they represent the two extremes in the spectrum of consistency models. Extension of the techniques for other models should be straightforward.

3 Prefetching

The previous section described how the delay constraints imposed by a consistency model limit the amount of buffering and pipelining among memory accesses. Prefetching provides one method for increasing performance by partially proceeding with an access that is delayed due to consistency model constraints. The following subsections describe the prefetch scheme proposed and provide insight into the strengths and weaknesses of the technique.

3.1 Description

Prefetching can be classified based on whether it is *binding* or *non-binding*, and whether it is controlled by *hardware* or *soft-*

ware. With a binding prefetch, the value of a later reference (e.g., a register load) is bound at the time the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the location during the interval between prefetch and reference. Hardware cache-coherent architectures, such as the Stanford DASH multiprocessor [18], can provide prefetching that is non-binding. With a non-binding prefetch, the data is brought close to the processor (e.g., into the cache) and is kept coherent until the processor actually reads the value. Thus, non-binding prefetching does not affect correctness for any of the consistency models and can be used as simply a performance boosting technique. The technique described in this section assumes *hardware-controlled non-binding prefetch*. We contrast this technique with previously proposed prefetch techniques in Section 6.

Prefetching can enhance performance by partially servicing large latency accesses that are delayed due to consistency model constraints. For a read operation, a *read prefetch* can be used to bring the data into the cache in a read-shared state while the operation is delayed due to consistency constraints. Since the prefetch is non-binding, we are guaranteed that the read operation will return a correct value once it is allowed to perform, regardless of when the prefetch completed. In the majority of cases, we expect the result returned by the prefetch to be the correct result. The only time the result may be different is if the location is written to between the time the prefetch returns the value and the time the read is allowed to perform. In this case, the prefetched location would either get invalidated or updated, depending on the coherence scheme. If invalidated, the read operation will miss in the cache and access the new value from the memory system, as if the prefetch never occurred. In the case of an update protocol, the location is kept up-to-date, thus providing the new value to the read operation.

For a write operation, a *read-exclusive prefetch* can be used to acquire exclusive ownership of the line, enabling the write to that location to complete quickly once it is allowed to perform. A read-exclusive prefetch is only possible if the coherence scheme is invalidation-based. Similar to the read prefetch case, the line is invalidated if another processor writes to the location between the time the read-exclusive prefetch completes and the actual write operation is allowed to proceed. In addition, exclusive ownership is surrendered if another processor reads the location during that time.

3.2 Implementation

This subsection discusses the requirements that the prefetch technique imposes on a multiprocessor architecture. Let us first consider how the proposed prefetch technique can be incorporated into the processor environment. Assume the general case where the processor has a load and a store buffer. The usual way to enforce a consistency model is to delay the issue of accesses in the buffer until certain previous accesses complete. Prefetching can be incorporated in this framework by having the hardware automatically issue a prefetch (read prefetch for reads and read-exclusive prefetch for writes and atomic read-modify-writes) for accesses that are in the load or store buffer, but are delayed due to consistency constraints. A prefetch buffer may be used to buffer multiple prefetch requests. Prefetches can be retired from this buffer as fast as the cache and memory system allow.

A prefetch request first checks the cache to see whether the line is already present. If so, the prefetch is discarded. Other-

wise, the prefetch is issued to the memory system. When the prefetch response returns to the processor, it is placed in the cache. If a processor references a location it has prefetched before the result has returned, the reference request is combined with the prefetch request so that a duplicate request is not sent out and the reference completes as soon as the prefetch result returns.

The prefetch technique discussed imposes several requirements on the memory system. Most importantly, the architecture requires hardware coherent caches. In addition, the location to be prefetched needs to be cachable. Also, to be effective for writes, prefetching requires an invalidation-based coherence scheme. In update-based schemes, it is difficult to partially service a write operation without making the new value available to other processors, which results in the write being performed.

For prefetching to be beneficial, the architecture needs a high-bandwidth pipelined memory system, including lockup-free caches [14, 21], to sustain several outstanding requests at a time. The cache will also be more busy since memory references that are prefetched access the cache twice, once for the prefetch and another time for the actual reference. As previously mentioned, accessing the cache for the prefetch request is desirable for avoiding extraneous traffic. We do not believe that the double access will be a major issue since prefetch requests are generated only when normal accesses are being delayed due to consistency constraints, and by definition, there are no other requests to the cache during that time.

Lookahead in the instruction stream is also beneficial for hardware-controlled prefetch schemes. Processors with dynamic instruction scheduling, whereby the decoding of an instruction is decoupled from the execution of the instruction, help in providing such lookahead. Branch prediction techniques that allow execution of instructions past unresolved conditional branches further enhance this. Aggressive lookahead provides the hardware with several memory requests that are being delayed in the load and store buffers due to consistency constraints (especially for the stricter memory consistency models) and gives prefetching the opportunity to pipeline such accesses. The strengths and weaknesses of hardware-controlled non-binding prefetching are discussed in the next subsection.

3.3 Strengths and Weaknesses

This subsection presents two example code segments to provide intuition for the circumstances where prefetching boosts performance and where prefetching fails. Figure 2 shows the two code segments. All accesses are to read-write shared locations. We assume a processor with non-blocking reads and branch prediction machinery. The cache coherence scheme is assumed to be invalidation-based, with cache hit latency of 1 cycle and cache miss latency of 100 cycles. Assume the memory system can accept an access on every cycle (e.g., caches are lockup-free). We also assume no other processes are writing to the locations used in the examples and that the lock synchronizations succeed (i.e., the lock is free).

Let us first consider the code segment on the left side of Figure 2. This code segment resembles a producer process updating the values of two memory locations. Given a system with sequential consistency, each access is delayed for the previous access to be performed. The first three accesses miss in the cache, while the unlock access hits due to the fact that exclusive ownership was gained by the previous lock access. Therefore, the four accesses take a total of 301 cycles to perform. In a system with release consistency, the write accesses are delayed until the

lock L	(miss)	lock L	(miss)
write A	(miss)	read C	(miss)
write B	(miss)	read D	(hit)
unlock L	(hit)	read E[D]	(miss)
		unlock L	(hit)

Example 1

Example 2

Figure 2: Example code segments.

lock access is performed, and the unlock access is delayed for the write accesses to perform. However, the write accesses are pipelined. Therefore, the accesses take 202 cycles.

The prefetch technique described in this section boosts the performance of both the sequential and release consistent systems. Concerning the loop that would be used to implement the lock synchronization, we assume the branch predictor takes the path that assumes the lock synchronization succeeds. Thus, the lookahead into the instruction stream allows locations A and B to be prefetched in read-exclusive mode. Regardless of the consistency model, the lock access is serviced in parallel with prefetch for the two write accesses. Once the result for the lock access returns, the two write accesses will be satisfied quickly since the locations are prefetched into the cache. Therefore, with prefetching, the accesses complete in 103 cycles for both SC and RC. For this example, prefetching boosts the performance of both SC and RC and also equalizes the performance of the two models.

We now consider the second code segment on the right side of Figure 2. This code segment resembles a consumer process reading several memory locations. There are three read accesses within the critical section. As shown, the read to location D is assumed to hit in the cache, and the read of array E depends on the value of D to access the appropriate element. For simplicity, we will ignore the delay due to address calculation for accessing the array element. Under SC, the accesses take 302 cycles to perform. Under RC, they take 203 cycles. With the prefetch technique, the accesses take 203 cycles under SC and 202 cycles under RC. Although the performance of both SC and RC are enhanced by prefetching, the maximum performance is not achieved for either model. The reason is simply because the address of the read access to array E depends on the value of D and although the read access to D is a cache hit, this access is not allowed to perform (i.e., the value can not be used by the processor) until the read of C completes (under SC) or until the lock access completes (under RC). Thus, while prefetching can boost performance by pipelining several accesses that are delayed due to consistency constraints, it fails to remedy the cases where out-of-order consumption of return values is important to allow the processor to proceed efficiently.

In summary, prefetching is an effective technique for pipelining large latency references even though the consistency model disallows it. However, prefetching fails to boost performance when out-of-order consumption of prefetched values is important. Such cases occur in many applications, where accesses that hit in the cache are dispersed among accesses that miss, and the out-of-order use of the values returned by cache hits is critical for achieving the highest performance. The next section describes a speculative technique that remedies this shortcoming by allowing the processor to consume return values out-of-order regardless of the consistency constraints. The combination of prefetching for stores and the speculative execution technique for loads will be shown to be effective in regaining opportunity for maximum pipelining and buffering.

4 Speculative Execution

This section describes the speculative execution technique for load accesses. An example implementation is presented in the latter part of the section. As will be seen, this technique is particularly applicable to superscalar designs that are being proposed for next generation microprocessors. Finally, the last subsection shows the execution of a simple code segment with speculative loads.

4.1 Description

The idea behind speculative execution is simple. Assume u and v are two accesses in program order, with u being any large latency access and v being a load access. In addition, assume that the consistency model requires the completion of v to be delayed until u completes. *Speculative execution for load accesses* works as follows. The processor obtains or assumes a return value for access v before u completes and proceeds. At the time u completes, if the return value for v used by the processor is the same as the current value of v , then the speculation is successful. Clearly, the computation is correct since even if v was delayed, the value the access returns would have been the same. However, if the current value of v is different from what was speculated by the processor, then the computation is incorrect. In this case, we need to throw out the computation that depended on the value of v and repeat that computation. The implementation of such a scheme requires a *speculation mechanism* to obtain a speculated value for the access, a *detection mechanism* to determine whether the speculation succeeded, and a *correction mechanism* to repeat the computation if the speculation was unsuccessful.

Let us consider the speculation mechanism first. The most reasonable thing to do is to perform the access and use the returned value. In case the access is a cache hit, the value will be obtained quickly. In the case of a cache miss, although the return value will not be obtained quickly, the access is effectively pipelined with previous accesses in a way similar to prefetching. In general, guessing on the value of the access is not beneficial unless the value is known to be constrained to a small set (e.g., lock accesses).

Regarding the detection mechanism, a naive way to detect an incorrect speculated value is to repeat the access when the consistency model would have allowed it to proceed under non-speculative circumstances and to check the return value with the speculated value. However, if the speculation mechanism performs the speculative access and keeps the location in the cache, it is possible to determine whether the speculated value is correct by simply monitoring the coherence transactions on that location. Thus, the speculative execution technique can be implemented such that the cache is accessed only once per access versus the two times required by the prefetch technique. Let us refer back to accesses u and v , where the consistency model requires the completion of load access v to be delayed until u completes. The speculative technique allows access v to be issued and the processor is allowed to proceed with the return value. The detection mechanism is as follows. An invalidation or update message for location v before u has completed indicates that the value of the access may be incorrect.² In addition, the lack of invalidation and update messages indicates

²There are two cases where the speculated value remains correct. The first is if the invalidation or update occurs due to false sharing, that is, for another location in the same cache line. The second is if the new value written is the same as the speculated value. We conservatively assume the speculated value is incorrect in either case.

that the speculated value is correct. Cache replacements need to be handled properly also. If location v is replaced from the cache before u completes, then invalidation and update messages may no longer reach the cache. The speculated value for v is assumed stale in such a case (unless one is willing to repeat the access once u completes and to check the current value with the speculated value). The next subsection provides further implementation details for this mechanism.

Once the speculated value is determined to be wrong, the correction mechanism involves discarding the computation that depended on the speculated value and repeating the access and computation. This mechanism is almost the same as the correction mechanism used in processors with branch prediction machinery and the ability to execute instructions past unresolved branches. With branch prediction, if the prediction is determined to be incorrect, the instructions and computation following the branch are discarded and the new target instructions are fetched. In a similar way, if a speculated value is determined to be incorrect, the load access and the computation following it can be discarded and the instructions can be fetched and executed again to achieve correctness.

The speculative technique overcomes the shortcoming of the prefetch technique by allowing out-of-order consumption of speculated values. Referring back to the second example in Figure 2, let us consider how well the speculative technique performs. We still assume that no other processes are writing to the locations. Speculative execution achieves the same level of pipelining achieved by prefetching. In addition, the read access to D no longer hinders the performance since its return value is allowed to be consumed while previous accesses are outstanding. Thus, both SC and RC complete the accesses in 104 cycles.

Given speculative execution, load accesses can be issued as soon as the address for the access is known, regardless of the consistency model supported. Similar to the prefetch technique, the speculative execution technique imposes several requirements on the memory system. Hardware-coherent caches are required for providing an efficient detection mechanism. In addition, a high-bandwidth pipelined memory system with lockup-free caches [14, 21] is necessary to sustain multiple outstanding requests.

4.2 Example Implementation

This subsection provides an example implementation of the speculative technique. We use a processor that has dynamic scheduling and branch prediction capability. As with prefetching, the speculative technique also benefits from the lookahead in the instruction stream provided by such processors. In addition, the correction mechanism for the branch prediction machinery can easily be extended to handle correction for speculative load accesses. Although such processors are complex, incorporating speculative execution for load accesses into the design is simple and does not significantly add to the complexity. This subsection begins with a description of a dynamically scheduled processor that we chose as the base for our example implementation. Next, the details of implementing speculative execution for load accesses are discussed.

We obtained the organization for the base processor directly from a study by Johnson [11] (the organization is also described by Smith et al. [23] as the MATCH architecture). Figure 3 shows the overall structure of the processor. Only a brief description of the processor is given; the interested reader is referred to Johnson's thesis [11] for more detail. The processor

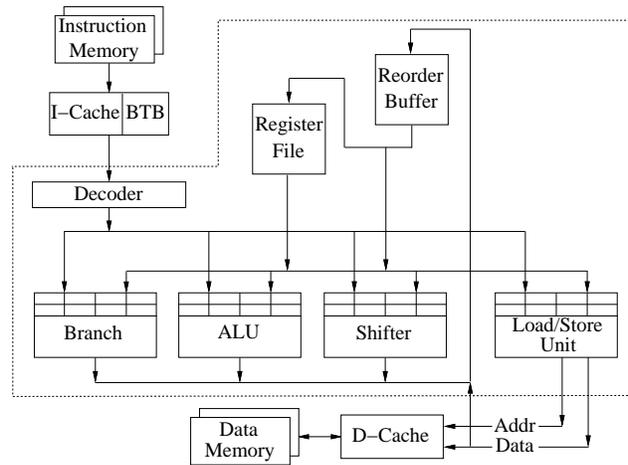


Figure 3: Overall structure of Johnson's dynamically scheduled processor.

consists of several independent function units. Each functional unit has a *reservation station* [25]. The reservation stations are instruction buffers that decouple instruction decoding from the instruction execution and allow for dynamic scheduling of instructions. Thus, the processor can execute instructions out of order though the instructions are fetched and decoded in program order. In addition, the processor allows execution of instructions past unresolved conditional branches. A branch target buffer (BTB) [16] is incorporated into the instruction cache to provide conditional branch prediction.

The *reorder buffer* [22] used in the architecture is responsible for several functions. The first function is to eliminate storage conflicts through register renaming [12]. The buffer provides the extra storage necessary to implement register renaming. Each instruction that is decoded is dynamically allocated a location in the reorder buffer and a tag is associated with its result register. The tag is updated to the actual result value once the instruction completes. When a later instruction attempts to read the register, correct execution is achieved by providing the value (or tag) in the reorder buffer instead of the value in the register file. Unresolved operand tags in the reservation stations are also updated with the appropriate value when the instruction with the corresponding result register completes.

The second function of the reorder buffer is to allow the processor to execute instructions past unresolved conditional branches by providing storage for the uncommitted results. The reorder buffer functions as a FIFO queue for instructions that have not been committed. When an instruction at the head of the queue completes, the location belonging to it is deallocated and the result value is written to the register file. Since the processor decodes and allocates instructions in program order, the updates to the register file take place in program order. Since instructions are kept in FIFO order, instructions in the reorder buffer that are ahead of a branch do not depend on the branch, while the instructions after the branch are control dependent on it. Thus, the results of instructions that depend on the branch are not committed to the register file until the branch completes. In addition, memory stores that are control dependent on the conditional branch are held back until the branch completes. If the branch is mispredicted, all instructions that are after the branch are invalidated from the reorder buffer, the reservation stations and buffers are appropriately cleared, and decoding and execution is started from the correct branch target.

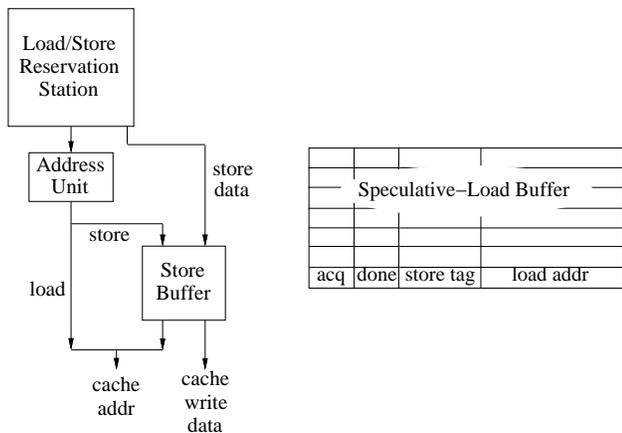


Figure 4: Organization of the load/store functional unit.

The mechanism provided in the reorder buffer for handling branches is also used to provide precise interrupts. Precise interrupts are provided to allow the processor to restart quickly without need to save and restore a lot of state. We will discuss the effect of requiring precise interrupts on the implementation of consistency models later in the section. Thus the reorder buffer plays an important role by eliminating storage conflicts through register renaming, allowing conditional branches to be bypassed, and providing precise interrupts.

To implement the speculative load technique, only the load/store (memory) unit of the base processor needs to be modified and the rest of the components remain virtually unchanged. Figure 4 shows the components of the memory unit. We first describe the components shown on the left side of the figure. These components are present regardless of whether speculative loads are supported. The only new component that is required for supporting speculative loads is the *speculative-load buffer* that will be described later.

The *load/store reservation station* holds decoded load and store instructions in program order. These instructions are retired to the address unit in a FIFO manner. Since the effective address for the memory instruction may depend on an unresolved operand, it is possible that the address for the instruction at the head of the reservation station can not be computed. The retiring of instructions is stalled until the effective address for the instruction at the head can be computed. The *address unit* is responsible for computing the effective address and doing the virtual to physical translation. Once the physical address is obtained, the address and data for store operations are placed into the *store buffer*. The retiring of stores from the store buffer is done in a FIFO manner and is controlled by the reorder buffer to assure precise interrupts (the mechanism is explained in the next paragraph). Load operations are allowed to bypass the store buffer and dependence checking is done on the store buffer to assure a correct return value for the load. Although the above implementation is sufficient for a uniprocessor, we need to add mechanisms to enforce consistency constraints for a multiprocessor.

Let us first consider how access order can be guaranteed for sequential consistency. The conventional method for achieving this is to delay the completion of each access until its previous access is complete. We first consider how the store operations are delayed appropriately. In general, a store operation may need to be delayed until certain previous load or store operations are completed. The mechanism for delaying the store is aided by the

fact that stores are already withheld to provide precise interrupts. The mechanism is as follows. All uncommitted instructions are allocated a location in the reorder buffer and are retired in program order. Except for a store instruction, an instruction at the head of the reorder buffer is retired when it completes. For store instructions, the store is retired from the reorder buffer as soon as the address translation is done. The reorder buffer controls the store buffer by signaling when it is safe to issue a store to the memory system. This signal is given when the store reaches the head of the reorder buffer. Consequently, a store is not issued until all previous loads and computation are complete. This mechanism satisfies the requirements placed by the SC model on a store with respect to previous loads. To make the implementation simpler for SC, we change the policy for retiring stores such that the store at the head of the reorder buffer is not retired until it completes also (for RC, however, the store at the head is still retired as soon as address translation is done). Thus, under SC, the store is also delayed for previous stores to complete and the store buffer ends up issuing stores one-at-a-time.

We now turn to how the restrictions on load accesses are satisfied. First, we discuss the requirements assuming the speculative load mechanism is not used. For SC, it is sufficient to delay a load until previous loads and stores have completed. This can be done by stalling the load/store reservation station at loads until the previous load is performed and the store buffer empties.

For speculative execution of load accesses, the mechanism for satisfying the restrictions on load accesses is changed. The major component for supporting the mechanism is the speculative-load buffer. The reservation station is no longer responsible for delaying certain load accesses to satisfy consistency constraints. A load is issued as soon as its effective address is computed. The speculation mechanism comprises of issuing the load as soon as possible and using the speculated result when it returns.

The speculative-load buffer provides the detection mechanism by signaling when the speculated result is incorrect. The buffer works as follows. Loads that are retired from the reservation station are put into the buffer in addition to being issued to the memory system. There are four fields per entry (as shown in Figure 4): load address, acq, done, and store tag. The load address field holds the physical address for the load. The acq field is set if the load is considered an acquire access. For SC, all loads are treated as acquires. The done field is set when the load is performed. If the consistency constraints require the load to be delayed for a previous store, the store tag uniquely identifies that store. A null store tag specifies that the load depends on no previous stores. When a store completes, its corresponding tag in the speculative-load buffer is nullified if present. Entries are retired in a FIFO manner. Two conditions need to be satisfied before an entry at the head of the buffer is retired. First, the store tag field should equal null. Second, the done field should be set if the acq field is set. Therefore, for SC, an entry remains in the buffer until all previous load and store accesses complete and the load access it refers to completes. Appendix A describes how an atomic read-modify-write can be incorporated in the above implementation.

We now describe the detection mechanism. The following coherence transactions are monitored by the speculative-load buffer: invalidations (or ownership requests), updates, and replacements.³ The load addresses in the buffer are associatively

³A replacement is required if the processor accesses an address that maps onto a cache line with valid data for a different address. To avoid deadlock, a replacement request to a line with an outstanding access needs to be delayed until the access completes.

checked for a match with the address of such transactions.⁴ Multiple matches are possible. We assume the match closest to the head of the buffer is reported. A match in the buffer for an address that is being invalidated or updated signals the possibility of an incorrect speculation. A match for an address that is being replaced signifies that future coherence transactions for that address will not be sent to the processor. In either case, the speculated value for the load is assumed to be incorrect.

Guaranteeing the constraints for release consistency can be done in a similar way to SC. The conventional way to provide RC is to delay a release access until its previous accesses complete and to delay accesses following an acquire until the acquire completes. Let us first consider delays for stores. The mechanism that provides precise interrupts by holding back store accesses in the store buffer is sufficient for guaranteeing that stores are delayed for the previous acquire. Although the mechanism described is stricter than what RC requires, the conservative implementation is required for providing precise interrupts. The same mechanism also guarantees that a release (which is simply a special store access) is delayed for previous load accesses. To guarantee a release is also delayed for previous store accesses, the store buffer delays the issue of the release operation until all previously issued stores are complete. In contrast to SC, however, ordinary stores are issued in a pipelined manner.

Let us consider the restriction on load accesses under RC. The conventional method involves delaying a load access until the previous acquire access is complete. This can be done by stalling the load/store reservation station after an acquire access until the acquire completes. However, the reservation station need not be stalled if we use the speculative load technique. Similar to the implementation of SC, loads are issued as soon as the address is known and the speculative-load buffer is responsible for detecting incorrect values. The speculative-load buffer description given for SC applies for RC. The only difference is that the *acq* field is only set for accesses that are considered acquire accesses under RC. Therefore, for RC, an entry remains in the speculative-load buffer until all previous acquires are completed. Furthermore, an acquire entry remains in the buffer until it completes also. The detection mechanism described for SC remains unchanged.

When the speculative-load buffer signals an incorrect speculated value, all computation that depends on that value needs to be discarded and repeated. There are two cases to consider. The first case is that the coherence transaction (invalidation, update, or replacement) arrives after the speculative load has completed (i.e., *done* field is set). In this case, the speculated value may have been used by the instructions following the load. We conservatively assume that all instructions past the load instruction depend on the value of the load and the mechanism for handling branch misprediction is used to treat the load instruction as “mispredicted”. Thus, the reorder buffer discards the load and the instructions following it and the instructions are fetched and executed again. The second case occurs if the coherence transaction arrives before the speculative load has completed (i.e., *done* field is not set). In this case, only the speculative load needs to be reissued, since the instructions following it have not yet used an incorrect value. This can be done by simply reissuing the load access and does not require the instructions following the load to be discarded.⁵ The next subsection further

⁴It is possible to ignore the entry at the head of the buffer if the store tag is null. The null store tag for the head entry signifies that all previous accesses that are required to complete have completed and the consistency model constraints would have allowed the access to perform at such a time.

⁵To handle this case properly, we need to tag return values to distinguish between the initial return value, which has not yet reached the processor and

illustrates speculative loads by stepping through the execution of a simple code segment.

4.3 Illustrative Example

In this subsection, we step through the execution of the code segment shown at the top of Figure 5. The sequential consistency model is assumed. Both the speculative technique for loads and the prefetch technique for stores are employed. Figure 5 also shows the contents of several of the buffers during the execution. We show the detection and correction mechanism in action by assuming that the speculated value for location D (originally in the cache) is later invalidated.

The instructions are assumed to be decoded and placed in the reorder buffer. In addition, it is assumed that the load/store reservation station has issued the operations. The first event shows that both the loads and the exclusive prefetches for the stores have been issued. The store buffer is buffering the two store operations and the speculative-load buffer has entries for the three loads. Note that the speculated value for load D has already been consumed by the processor. The second event occurs when ownership arrives for location B. The completion of store B is delayed by the reorder buffer, however, since there is an uncommitted instruction ahead of the store (this observes precise interrupts). Event 3 signifies the arrival of the value for location A. The entry for load A is removed from the reorder buffer and the speculative-load buffer since the access has completed. Once load A completes, the store buffer is signaled by the reorder buffer to allow store B to proceed. Since location B is now cached in exclusive mode, store B completes quickly (event 4). Thus, store C reaches the head of the reorder buffer. The store buffer is signaled in turn to allow store C to issue and the access is merged with the previous exclusive prefetch request for the location.

At this point, we assume an invalidation arrives for location D. Since there is a match for this location in the speculation buffer and since the speculated value is used, the load D instruction and the following load instruction are discarded (event 5). Event 6 shows that these two instructions are fetched again and a speculative load is issued to location D. The load is still speculative since the previous store (store C) has not completed yet. Event 7 shows the arrival of the new value for location D. Since the value for D is known now, the load access E[D] can be issued. Note that although the access to D has completed, the entry remains in the reorder buffer since store C is still pending. Once the ownership for location C arrives (event 8), store C completes and is retired from the reorder buffer and the store buffer. In addition, load D is no longer considered a speculative load and is retired from both the reorder and the speculative-load buffers. The execution completes when the value for E[D] arrives (event 9).

5 Discussion

The two implementation techniques presented in this paper provide a greater opportunity for buffering and pipelining of accesses than any of the previously proposed techniques. This section discusses some implications associated with the two techniques.

needs to be discarded, and the return value from the repeated access, which is the one to be used. In addition, in case there are multiple matches for the address in the speculative-load buffer, we have to guarantee that initial return values are not used by any of the corresponding loads.

Example code segment:

```

read A      (miss)
write B     (miss)
write C     (miss)
read D      (hit)
read E[D]   (miss)

```

Event	Reorder Buffer	Store Buffer	Speculative-Load Buffer	Cache Contents
1 reads are issued and writes are prefetched	ld E[D] ld D st C st B ld A	st C st B	acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld E[D] ✓ <input type="checkbox"/> <input type="checkbox"/> st C ld D ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld A	A: ld pending B: ex-prf pending C: ex-prf pending D: valid E[D]: ld pending
2 ownership for B arrives	ld E[D] ld D st C st B ld A	st C st B	acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld E[D] ✓ <input type="checkbox"/> <input type="checkbox"/> st C ld D ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld A	A: ld pending B: valid exclusive C: ex-prf pending D: valid E[D]: ld pending
3 value for A arrives	ld E[D] ld D st C st B	st C st B	acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld E[D] ✓ <input type="checkbox"/> <input type="checkbox"/> st C ld D	A,D: valid B: valid exclusive C: ex-prf pending E[D]: ld pending
4 write to B completes	ld E[D] ld D st C	st C	acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld E[D] ✓ <input type="checkbox"/> <input type="checkbox"/> st C ld D	A,D: valid B: valid exclusive C: ex-prf pending E[D]: ld pending
5 invalidation for D arrives	st C	st C		A: valid B: valid exclusive C: ex-prf pending D: invalid
6 read of D is reissued	ld E[D] ld D st C	st C	acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld D	A: valid B: valid exclusive C: ex-prf pending D: ld pending
7 value for D arrives; read of E[D] is reissued	ld E[D] ld D st C	st C	acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld E[D] ✓ <input type="checkbox"/> <input type="checkbox"/> st C ld D	A,D: valid B: valid exclusive C: ex-prf pending E[D]: ld pending
8 ownership for C arrives	ld E[D]		acq done st tag ld addr ✓ <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> ld E[D]	A,D: valid B,C: valid exclusive E[D]: ld pending
9 value for E[D] arrives				A,D,E[D]: valid B,C: valid exclusive

Figure 5: Illustration of buffers during execution of the code segment.

The main idea behind the prefetch and speculative load techniques is to service accesses as soon as possible, regardless of the constraints imposed by the consistency model. Of course, since correctness needs to be maintained, the early service of the access is not always helpful. For these techniques to provide performance benefits, the probability that a prefetched or speculated value is invalidated must be small. There are several reasons why we expect such invalidations to be infrequent. Whether prefetched or speculated locations are invalidated loosely depends on whether it is critical to delay such accesses to obtain a correct execution. If the supported consistency model is a relaxed model such as RC, delays are imposed only at synchronization points. In many applications, the time at which one process releases a synchronization is long before the time another process tries to acquire the synchronization. This implies that no other process is simultaneously accessing the locations protected by the synchronization. Correctness can be achieved in this case without the need to delay the accesses following an acquire until the acquire completes or to delay a release until its previous accesses complete. For cases where the supported consistency model is strict, such as SC, the strict delays imposed on accesses are also rarely necessary for correctness. We have observed that most programs do not have data races and provide sequentially consistent results even when they are executed on a release consistent architecture (see [8] for a formal definition of programs with such a property). Therefore, most delays imposed on accesses by a typical SC implementation are superfluous for achieving correctness. It is important to

substantiate the above observations in the future with extensive simulation experiments.

A major implication of the techniques proposed is that the performance of different consistency models is equalized once these techniques are employed. If one is willing to implement these techniques, the choice of the consistency model to be supported in hardware becomes less important. Of course, the choice of the model for software can still affect optimizations, such as register allocation and loop transformation, that the compiler can perform on the program. However, this choice for software is orthogonal to the choice of the consistency model to support in hardware.

6 Related Work

In this section, we discuss previous work regarding prefetching and speculative execution. Next, we consider other proposed techniques for providing more efficient implementations of consistency models.

The main advantage of the prefetch scheme described in this study is that it is non-binding. Hardware-controlled binding prefetching has been studied by Lee [17]. Gornish, Granston, and Viedenbaum [10] have evaluated software-controlled binding prefetching. However, binding prefetching is quite limited in its ability to enhance the performance of consistency models. For example, in the SC implementation described, a binding prefetch can not be issued any earlier than the actual access is allowed to be issued.

Non-binding prefetching is possible if hardware cache-coherence is provided. Software-controlled non-binding prefetching has been studied by Porterfield [20], Mowry and Gupta [19], and Gharachorloo et al. [7]. Porterfield studied the technique in the context of uniprocessors while the work by Mowry and Gharachorloo studies the effect of prefetching for multiprocessors. In [7], we provide simulation results, for processors with blocking reads, that show software-controlled prefetching boosts the performance of consistency models and diminishes the performance difference among models if both read and read-exclusive prefetches are employed. Unfortunately, software-controlled prefetching requires the programmer or the compiler to anticipate accesses that may benefit from prefetching and to add the appropriate instructions in the application to issue a prefetch for the location. The advantage of hardware-controlled prefetching is that it does not require software help, neither does it consume instructions or processor cycles to do the prefetch. The disadvantage of hardware-controlled prefetching is that the prefetching window is limited to the size of the instruction lookahead buffer, while theoretically, software-controlled non-binding prefetching has an arbitrarily large window. In general, it should be possible to combine hardware-controlled and software-controlled non-binding prefetching such that they complement one another.

Speculative execution based on possibly incorrect data values has been previously described by Tom Knight [13] in the context of providing dynamic parallelism for sequential Lisp programs. The compiler is assumed to transform the program into sequences of instructions called transaction blocks. These blocks are executed in parallel and instructions that side-effect main memory are delayed until the block is committed. The sequential order of blocks determines the order in which the blocks can commit their side-effects. The mechanism described by Knight checks to see whether a later block used a value that is changed by the side-effecting instructions of the current block, and in case of a conflict, aborts the execution of the later block.

This is safe since the side-effecting instructions of later blocks are delayed, making the blocks fully restartable. Although this scheme is loosely similar to the speculative load scheme discussed in this paper, our scheme is unique in the way speculation is used to enhance the performance of multiprocessors given a set of consistency constraints.

Adve and Hill [1] have proposed an implementation for sequential consistency that is more efficient than conventional implementations. Their scheme requires an invalidation-based cache coherence protocol. At points where a conventional implementation stalls for the full latency of pending writes, their implementation stalls only until ownership is gained. To make the implementation satisfy sequential consistency, the new value written is not made visible to other processors until all previous writes by this processor have completed. The gains from this are expected to be limited, however, since the latency of obtaining ownership is often only slightly smaller than the latency for the write to complete. In addition, the proposed scheme has no provision for hiding the latency of read accesses. Since the visibility-control mechanism reduces the stall time for writes only slightly and does not affect the stall time for reads, we do not expect it to perform much better than conventional implementations. In contrast, the prefetch and speculative load techniques provide much greater opportunity for buffering and pipelining of read and write accesses.

Stenstrom [24] has proposed a mechanism for guaranteeing access order at the memory instead of at the processor. Each request contains a processor identification and a sequence number. Consecutive requests from the same processor get consecutive sequence numbers. Each memory module has access to a common data structure called *next sequence-number table* (NST). The NST contains P entries, one entry per processor. Each entry contains the sequence number of the next request to be performed by the corresponding processor. This allows the mechanism to guarantee that accesses from each processor are kept in program order. Theoretically, this scheme can enhance the performance of sequential consistency. However, the major disadvantage is that caches are not allowed. This can severely hinder the performance when compared to implementations that allow shared locations to be cached. Furthermore, the technique is not scalable to a large number of processors since the *increment network* used to update the different NST's grows quadratically in connections as the number of processors increases.

The detection mechanism described in Section 4 is interesting since it can be extended to detect violations of sequential consistency in architectures that implement more relaxed models such as release consistency. Release consistent architectures are guaranteed to provide sequential consistency for programs that are free of data races [8]. However, determining whether a program is free of data races is undecidable [3] and is left up to the programmer. In [6], we present an extension of the detection mechanism which, for every execution of the program, determines *either* that the execution is sequentially consistent *or* that the program has data races and may result in sequentially inconsistent executions. Since it is not desirable to conservatively detect violations of SC for programs that are free of data races and due to the absence of a correction mechanism, the detection technique used in [6] is less conservative than the one described here. In addition, the extended technique needs to check for violations of SC arising from performing either a read or a write access out of order.

7 Concluding Remarks

To achieve higher performance, a number of relaxed memory consistency models have been proposed for shared memory multiprocessors. Unfortunately, the relaxed models present a more complex programming model. In this paper, we have proposed two techniques, that of prefetching and speculative execution, that boost the performance under all consistency models. The techniques are also noteworthy in that they allow the strict models, such as sequential consistency, to achieve performance close to that of relaxed models like release consistency. The cost, of course, is the extra hardware complexity associated with the implementation of the techniques. While the prefetch technique is simple to incorporate into cache-coherent multiprocessors, the speculative execution technique requires more sophisticated hardware support. However, the mechanisms required to implement speculative execution are similar to those employed in several next generation superscalar processors. In particular, we showed how the speculative technique could be incorporated into one such processor design with minimal additional hardware.

8 Acknowledgments

We greatly appreciate the insight provided by Ruby Lee that led to the prefetch technique described in this paper. We thank Mike Smith and Mike Johnson for helping us with the details of out-of-order issue processors. Sarita Adve, Phillip Gibbons, Todd Mowry, and Per Stenstrom provided useful comments on an earlier version of this paper. This research was supported by DARPA contract N00014-87-K-0828. Kourosh Gharachorloo is partly supported by Texas Instruments. Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award with matching funds from Sumitomo, Tandem, and TRW.

References

- [1] Sarita Adve and Mark Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I: 47–50, August 1990.
- [2] Sarita Adve and Mark Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.
- [3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.
- [4] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.
- [5] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.
- [6] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting violations of sequential consistency. In *Symposium on Parallel Algorithms and Architectures*, July 1991.
- [7] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models

- for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.
- [8] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
- [9] James R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.
- [10] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, September 1990.
- [11] William M. Johnson. *Super-Scalar Processor Design*. PhD thesis, Stanford University, June 1989.
- [12] R. M. Keller. Look-ahead processors. *Computing Surveys*, 7(4):177–195, 1975.
- [13] Tom Knight. An architecture for mostly functional languages. In *ACM Conference on Lisp and Functional Programming*, 1986.
- [14] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.
- [15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.
- [16] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17:6–22, 1984.
- [17] Roland Lun Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, May 1987.
- [18] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
- [19] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, June 1991.
- [20] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.
- [21] Christoph Scheurich and Michel Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages I: 118–125, August 1988.
- [22] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.
- [23] Michael Smith, Monica Lam, and Mark Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, May 1990.
- [24] Per Stenstrom. A latency-hiding access ordering scheme for multiprocessors with buffered multistage networks. Technical Report Department of Computer Engineering, Lund University, Sweden, November 1990.
- [25] R. M. Tomasulo. An efficient hardware algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, 1967.

Appendix A: Read-Modify-Write Accesses

Atomic read-modify-write accesses are treated differently from other accesses as far as speculative execution is concerned. Some read-modify-write locations may not be cached. The simplest way to handle such locations is to delay the access until previous accesses that are required to complete by the consistency model have completed. Thus, there is no speculative load for non-cached read-modify-write accesses. In the following, we will describe the treatment of cachable read-modify-write accesses under the SC model. Extensions for the RC model are straightforward.

The reorder buffer treats the read-modify-writes as both a read and a write. Therefore, as is the case for normal reads, the read-modify-write retires from the head of the reorder buffer only when the access completes. In addition, similar to a normal write, the reorder buffer signals the store buffer when the read-modify-write reaches the head. The load/store reservation station services a read-modify-write by splitting it into two operations, a speculative load that results in a read-exclusive request and the actual atomic read-modify-write. The speculative load is issued to the memory system and is also placed in the speculative-load buffer. The read-modify-write is simply placed in the store buffer. The store tag for the speculative load entry is set to the tag for the read-modify-write in the store buffer. The done field is set when exclusive ownership is attained for the location and the return value is sent to the processor. The actual read-modify-write occurs when the read-modify-write is issued by the store buffer. The entry corresponding to the speculative load is guaranteed to be at the head of the speculative-load buffer when the actual read-modify-write is issued. In the case of a match on the speculative load entry before the read-modify-write is issued, the read-modify-write and the computations following it are discarded from all buffers and are repeated. The return result of the speculative read-exclusive request is ignored if it reaches the processor after the read-modify-write is issued by the store buffer (the processor simply waits for the return result of the read-modify-write). If a match on the speculative load entry occurs after the read-modify-write is issued, only the computation following the read-modify-write is discarded and repeated. In this case, the value used for the read-modify-write will be the return value of the issued atomic access. The speculative load entry is retired from the speculative buffer when the read-modify-write completes.