Architectural Support for Relaxed Memory Consistency Models

Brian Grayson and Craig Chase

TR TR-PDS-1998-007 July 1998

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Architectural Support for Relaxed Memory Consistency Models
Technical Report TR-PDS-1998-007

Brian Grayson* bgrayson@ece.utexas.edu

Craig Chase chase@ece.utexas.edu

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084

July 1998

Keywords: memory access ordering, instruction-level-parallelism, memory consistency models, synchronization, shared-memory

Abstract

Shared-memory multiprocessors frequently allow reordering of memory accesses to help hide the latency of remote accesses. A "memory consistency model" describes which reorderings of memory accesses are allowed. Researchers have investigated several memory consistency models that provide a high degree of latency tolerance around synchronization operations, which typically incur the highest latency. However, most processors implement stronger models. This paper presents a set of techniques for instruction ordering that can be used to provide the processor with the ability to implement a highly aggressive Release Consistency model. The same techniques can be used to provide support for even weaker memory models, such as allowing two critical sections to be performed out-of-order. Our analysis shows that the biggest impediment to instruction-level parallelism around synchronization operations is the spin-loop inherent in a traditional lock acquisition. Weakening the memory model

*Research supported in part by a National Science Foundation Graduate Fellowship.

from a typical Release Consistency model to a very aggressive Release Consistency model provided a smaller but still significant improvement. Simulations using all of the techniques described in this paper show savings of dozens to hundreds of cycles per critical section for some SPLASH-2 benchmarks, and a speedup of 6 or more for a fine-grain microbenchmark.

1 Introduction

Over the past several years, numerous memory consistency models [15, 10, 7, 3, 9, 4, 13, 5, 8] have been introduced to allow shared-memory multiprocessors to exploit higher degrees of memory-access parallelism. One of the most influential of the weaker memory models is Release Consistency (RC) [9]. The main advantage of Release Consistency over stronger memory models such as Weak Consistency [7] is that RC allows some amount of overlap between a protected critical section and surrounding code, and even allows two consecutive critical sections to have their execution partially overlapped.

Unfortunately, most processors actually implement models stronger than RC. For example, most implementations use a mechanism like a `Test&Test&Set` loop followed by a memory barrier to provide the acquire functionality. The memory barrier explicitly delays the critical section until the acquire and all earlier accesses have globally performed. Release Consistency is less strict — the acquire operation (but not necessarily all previous accesses) must be globally performed before the accesses in the critical section can be performed. A similar strengthening with respect to RC occurs due to the memory barrier before a release operation. To our knowledge, no benchmark results have been reported for a very aggressive RC implementation.

We address the gap between an aggressive RC model and existing processors by presenting some new architectural techniques, along with modifications to previous techniques, to provide efficient support for very weak memory consistency models. Our focus is on small-scale (4 to 16 node) hardware shared memory systems, where the expected latency to acquire a lock is usually in the range of tens to hundreds of cycles. We provide simulation results that show that, even on applications in which there is a very limited amount of instruction-level parallelism (ILP) near critical sections, these techniques are still able to save on average dozens or more cycles of execution time at **each** critical section.

2 Motivation

The *memory consistency model* of a shared-memory system dictates which reorderings of local memory operations are allowed — when certain local memory operations must be globally visible before other subsequent instructions can be executed. For example, a strong memory consistency model may not allow any reorderings, while a weaker memory model may allow writes to be performed out of order.

Although the ordering from strongest to weakest of memory consistency models is in general a partial order, most of the commonly-discussed models can be ordered with respect to each other. In the following (incomplete) list, memory models are listed in order from strongest (most restrictive) to weakest (least restrictive/best performance): linearizability [12], Sequential Consistency (SC) [15], Processor Consistency (PC) [10], Weak Consistency (WC) [7], Release Consistency (RC) [9], Scope Consistency (ScC) [13], Entry Consistency (EC) [5], and Location Consistency [8]. The interested reader should examine these papers, and some of the tutorial papers such as [2, 17], for further details.

In general, the models weaker than Processor Consistency allow reads and writes to be reordered, except near synchronization operations. For Weak Consistency, no synchronization operation may be performed until all previous accesses are performed, and no subsequent accesses may be performed until the synchronization completes. Release Consistency (RC) [9] distinguishes the synchronization operations into acquires, which are used to enter a critical section, and releases, which are used to leave a critical section. Release Consistency provides two specific "relaxations" with respect to the stronger models:

- Accesses *preceding* an acquire can be performed after the acquire.
- Accesses *following* a release can be performed before the release.

One interesting consequence of these relaxations, as mentioned in [9], is that if an instruction stream contains two non-nested critical sections which are protected by different locks, RC allows the acquire operation from the second critical section to be initiated early, perhaps as early as immediately after the first acquire completes. In addition, if the second acquire completes before the first critical section's release, any operations from the second critical section (except the release) may also be performed early, potentially allowing a two-fold reduction in the execution time of this portion of the code stream.

As an example, consider the situation shown in Figure 1. Under stronger models, the execution time for the two critical sections would be 190 cycles, while under an aggressive implementation of RC the second acquire and critical section can be completely overlapped with the 100 cycles of the first critical section, for an overall running time of 130 cycles.

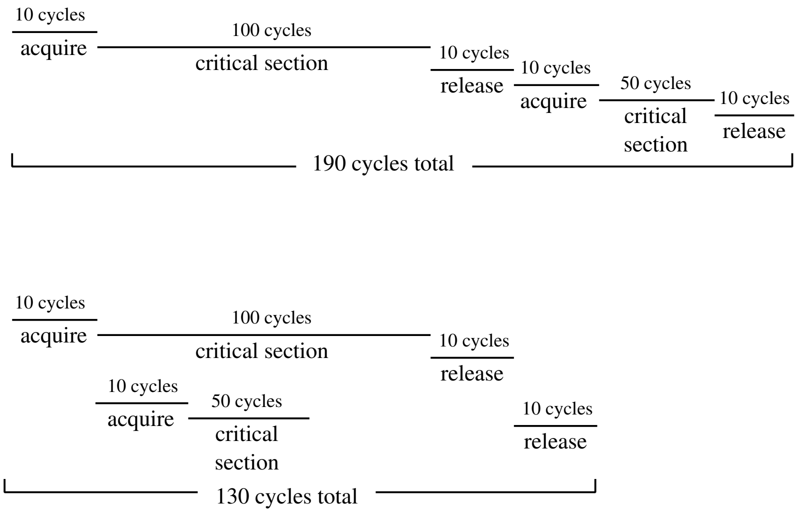

Figure 1: Overlapping critical sections under Release Consistency

To expose such a high degree of ILP to the processor, then, a system with an aggressive RC memory model needs to provide acquire and release primitives that meet the requirements of RC without imposing additional "strengthenings" of the memory model via impediments to memory access parallelism. In particular, such a system needs an efficient method of acquiring a lock without conditional branches, which impair instruction lookahead. The system also needs to provide several specific memory barriers near acquires and releases:

one-to-all barrier: all future accesses are delayed until the acquire is performed globally.

all-to-one barrier: the release is delayed until all previous accesses are performed globally.

Note that a traditional all-to-all barrier can be implemented by an all-to-one barrier followed immediately by a one-to-all barrier, provided there is some mechanism for ensuring the second barrier is executed after the first.

If one wished to provide support for weaker memory models (such as Scope Consistency (ScC) [13] or Entry Consistency (EC) [5]), then the system should also provide finer-grained *partial barriers*, described here in terms of critical sections:

one-to-some barrier: accesses within the critical section are delayed until the acquire completes (but accesses after the release may be performed before the acquire).

some-to-one barrier: the release is delayed until all accesses from within the critical section complete (but accesses before the acquire can be performed after the release).

In particular, the support for these fine-grained barriers should not add ordering restrictions on operations before and after the critical sections. Thus, an access before a critical section can be (globally) performed late, after the critical section operations have all finished; and an access after the critical section can be (globally) performed early, perhaps before the acquire completes or is even attempted.

As stated earlier, most commodity processors provide an all-to-all memory barrier. Some processors support a slightly weaker barrier (the PowerPC [18] and Sparc V9 [21] architectures, for example) but do not support a mechanism weak enough to support a very aggressive RC rather than a stronger RC model.

3 Related Work

Other researchers have attacked portions of the problem of providing support for aggressive RC. For example, the original RC paper [9] discussed an "immediate fence" operation, which delays the execution of the next access until previous accesses have completed, thus providing the all-to-one barrier needed before a release operation. However, Gharachorloo *et al.* [9] mention that on the DASH machine write fences were used instead of immediate fences. We are aware of only one system that provides an immediate fence operation: the simulator used for QOLB studies [14], which also currently assumes blocking loads, and thus does not support the most aggressive RC memory consistency model.

Pai *et al.* [20] discussed several techniques for increasing ILP around synchronization operations. The technique of *selective acquires* is used to provide support for the one-to-all barrier. In selective acquires, the programmer or compiler uses the return value from the acquire operation (such as the value from the successful `Test&Test&Set`) to "modify"[1] the pointers to all data accessed within the critical section. While this technique provides the required ordering, one side effect is to create a large number of accesses with addresses that will not be known until the acquire operation completes, which will prevent any later independent accesses from being performed due to the memory disambiguation problem. Another downside to this technique, besides the extra instructions for the pointer manipulations, is that it requires nontrivial programmer/compiler effort to make sure that **every** access within the critical section is properly modified: If the critical section calls a subroutine, all code in the subroutine (and any child subroutines) will need to be modified to perform these pointer-manipulations. We independently developed a more general technique, *synthetic dependences* (discussed in the next section), which, in combination with the other techniques described in this paper, can be used to provide such lightweight memory-access orderings without suffering from the above problems.

Also in [20], Pai *et al.* suggested a hardware construct called a *synchronization buffer*, placed in the memory hierarchy, which removes the polling/conditional branches associated with a traditional `Test&Test&Set` acquire implementation. When the synchronization buffer sees an acquire operation, it attempts to acquire the specified lock. If it fails, it continues retrying, in hardware, until it succeeds. By removing the conditional branches/control-flow dependences from the acquire operation, the processor is able to look further ahead and perform better scheduling of the instructions within the critical section and beyond, and can actually non-speculatively execute instructions such as arithmetic operations that do not directly or indirectly depend on the acquire instruction. One point worth mentioning is that the synchronization buffer does not reduce bus traffic: it will cause a bus operation for each successful **and unsuccessful** acquire attempt.

Goodman *et al.* proposed the *QOLB* [11] (pronounced "colby") bus transaction in order to allow for further overlap of computation and communication around lock operations. In particular, the QOLB technique keeps a (distributed) linked list of nodes waiting to acquire the lock, and ownership of the lock's line is passed directly to the next node on the list. This avoids the flurry of bus action seen when

[1] One would usually take the return value, AND it with 0, and then add this 0 value to the pointers to create a dataflow dependence without actually modifying the effective address.

a lock is released and the other nodes all attempt to acquire it.

4 Techniques

There are two major impediments to instruction-level parallelism near critical-section synchronization operations: most processors do not provide fine-grained memory ordering support, and most processors do not provide blocking atomic operations (thus requiring a check for success, and looping in the case of failure). We present several techniques that, when used together, allow much higher instruction-level parallelism around these synchronizations.

4.1 Synthetic Dependences

The technique of *synthetic dependences* can be considered a generalization of *selective acquires* [20]: *synthetic dependences* are an ordering requirement between one instruction and a later instruction represented through a register value or other ordering mechanism. In this section, we first discuss problems with selective acquires, and then discuss how to provide instruction set support for synthetic dependences without suffering from these problems.

As stated earlier, one problem with the technique of selective acquires is that it exacerbates the memory disambiguation problem. Processors must identify the read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) hazards between load and store instructions to the same address, and must ensure that the effect of these accesses is as if they were performed in program order. In addition, when an access' effective address is not yet known, the processor must behave conservatively, and thus enforce orderings between this access and all others until the address is known.

These memory disambiguation problems can be avoided by providing new load and store instructions that take additional general-purpose register operands: *load-dep* r_{addr}, $r_{SynDepIn}$, $r_{SynDepOut}$, r_{data} and *store-dep* r_{addr}, $r_{SynDepIn}$, $r_{SynDepOut}$, r_{data}. The access is not allowed to be performed until the values for all of its input registers, including the $r_{SynDepIn}$ register, are available. $r_{SynDepIn}$ thus allows us to delay the access instruction without affecting effective address calculation. The $r_{SynDepOut}$ register is written with a value of 0 when the instruction has been performed globally.

Notice that we can now enforce ordering between any two arbitrary accesses, without exacerbating the memory disambiguation problem — the first access should update a $r_{SynDepOut}$ register, and the second access should use this register as its $r_{SynDepIn}$ register parameter. An example of this is shown in Figure 2. In the code, a register operand that is not needed is given the pseudo-register mnemonic "nil."

```
store-dep loc, nil, dep1, val  #  Update location with val
store-dep flag, dep1, nil, 1   #  Set flag after update is complete
```

Figure 2: Example of load/store-with-dependence instructions

4.2 Partial Memory Barriers through Access Counting

Memory barrier instructions are needed near critical sections for two distinct purposes, as described earlier: to provide the one-to-all barrier for acquire, and the all-to-one barrier for releases. Most processors provide only an all-to-all barrier, which restricts possible reorderings. These ordering restrictions are shown graphically in Figure 3.

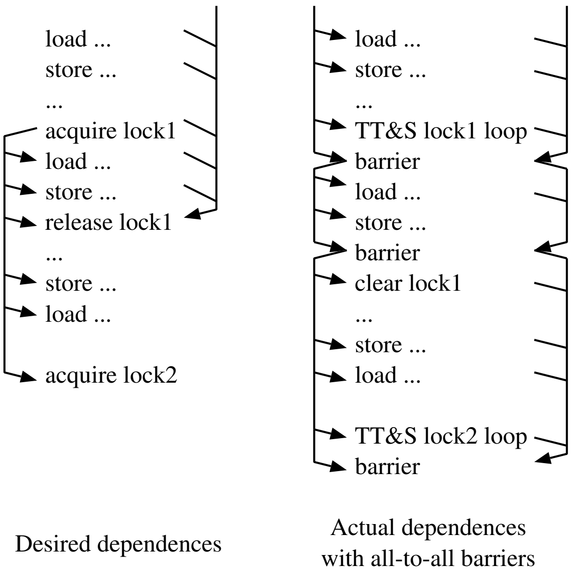

Figure 3: Dependences between memory accesses in an aggressive RC implementation and in a typical implementation.

Both one-to-some and some-to-one memory barriers can be provided through two new instructions: $CSBegin$ and $CSEnd$. $CSBegin$ marks the beginning of a critical section, and has two effects:

all subsequently-decoded instructions (until the corresponding $CSEnd$) are “marked,” and none of these instructions can execute until the $CSBegin$ completes (thus providing the one-to-some barrier). $CSEnd$ is the dual of $CSBegin$, and has two corresponding effects: all subsequently-decoded instructions are no longer marked, and the $CSEnd$ instruction will not execute until all of the marked instructions have completed (a some-to-one barrier).

In order to support more complicated synchronization activities such as nested critical sections, we allow each $CSBegin$ and $CSEnd$ instruction to take a 5-bit immediate value imm_5, specifying which $CSBegin$ should be paired with a given $CSEnd$. In addition, we frequently need to order a $CSBegin$ with respect to an early access (*i.e.* an acquire), and a release with respect to a $CSEnd$, so both of these new instructions also take synthetic dependence register parameters. Thus, the format of these instructions is $CSBegin\ imm5$, $r_{SynDepIn}$, $r_{SynDepOut}$ and $CSEnd\ imm5$, $r_{SynDepIn}$, $r_{SynDepOut}$.

Notice that support for these instructions is fairly easy to implement via *access counting*, similar to the fence counters mentioned in [9]. The processor keeps a counter for the memory barrier. When a $CSBegin$ is decoded, the decoder starts “marking” all subsequently decoded memory instructions (until the $CSEnd$). When a memory instruction is now decoded, it is marked and increments the counter, and when this instruction retires, it will decrement the counter (only marked memory instructions decrement the counter). A $CSEnd$ tells the decoder to stop marking future instructions, and also stalls until the counter again reaches 0. Thus, at any point, the counter contains the number of accesses within the critical section that have been decoded but not yet retired.

This technique can be extended to support nesting of critical sections through providing several counters, and changing the “marked” bit to an array of bits. In a nested situation, then, the memory instructions in the inner critical section would have two bits in their bit-masks set, and would increment two counters at decode time and decrement two counters at retirement.

These instructions can be used to prevent code within a critical section from being performed early, and the release from being performed early, as shown in Figure 4. Note that this example does not provide Release Consistency — as shown in Figure 5a, the partial barriers prescribe order relations only for instructions in the critical section, and allow accesses outside the critical section to be reordered with respect to the critical section. In order to satisfy the requirements of RC, all accesses after the acquire need to be delayed. We can provide this by beginning a critical section with $CSBegin$ after the acquire, and continuing this critical section until immediately after the next acquire. This is shown graphically in Figure 5b[2], where the acquire-to-subsequent-access dependences are shown to the left of the code. Similarly, we use another $CSBegin$/$CSEnd$ pair (counter 2) to delay the release until all accesses since the previous release have been globally performed. The access-to-subsequent-release dependences are shown to the right of the code in the figure. As shown in the figure, the use of partial barriers in this way provides exactly the ordering required for RC. This method was used in our simulations to support an aggressive RC memory model.

```
(acquire lock in some way, and provide synthetic dependence in cs_started)
CSBegin 0, nil, cs_started
(begin critical section operations)
...
(end critical section operations)
  #  Delay update of cs_done register until above accesses complete.
CSEnd 0, nil, cs_done
  #  Once the value for cs_done is available, reset the lock.
store-dep lock, cs_done, nil, 0
```

Figure 4: Example using $CSBegin$ and $CSEnd$. This code does not guarantee release consistency.

If counters are used to implement partial barriers, the implementation would undoubtedly benefit from "counter renaming" (similar in intent and design to register renaming). Our simulator provides 32 rename counters.

4.3 Blocking Acquire Instructions

Previous researchers have discussed optimizing acquire operations, including allowing prefetching of critical-section operands through collocation [14] (placing the data and its lock in the same cache line), removing the polling conditional branches through hardware support [20], and providing hardware queuing [14] to eliminate bus traffic and reduce latency. We combine these ideas by providing

[2]To avoid cluttering Figure 5b, we have removed the arcs from $CSBegin$ 2 to its subsequent marked instructions, and to $CSEnd$ 1 from its preceding marked instructions. These two instructions have nil for both of their synthetic dependence operands, and are used solely for marking the endpoints for the partial barriers. With counter renaming they can be executed early/late respectively, in which case the removed dependence arcs do not significantly affect the potential reorderings.

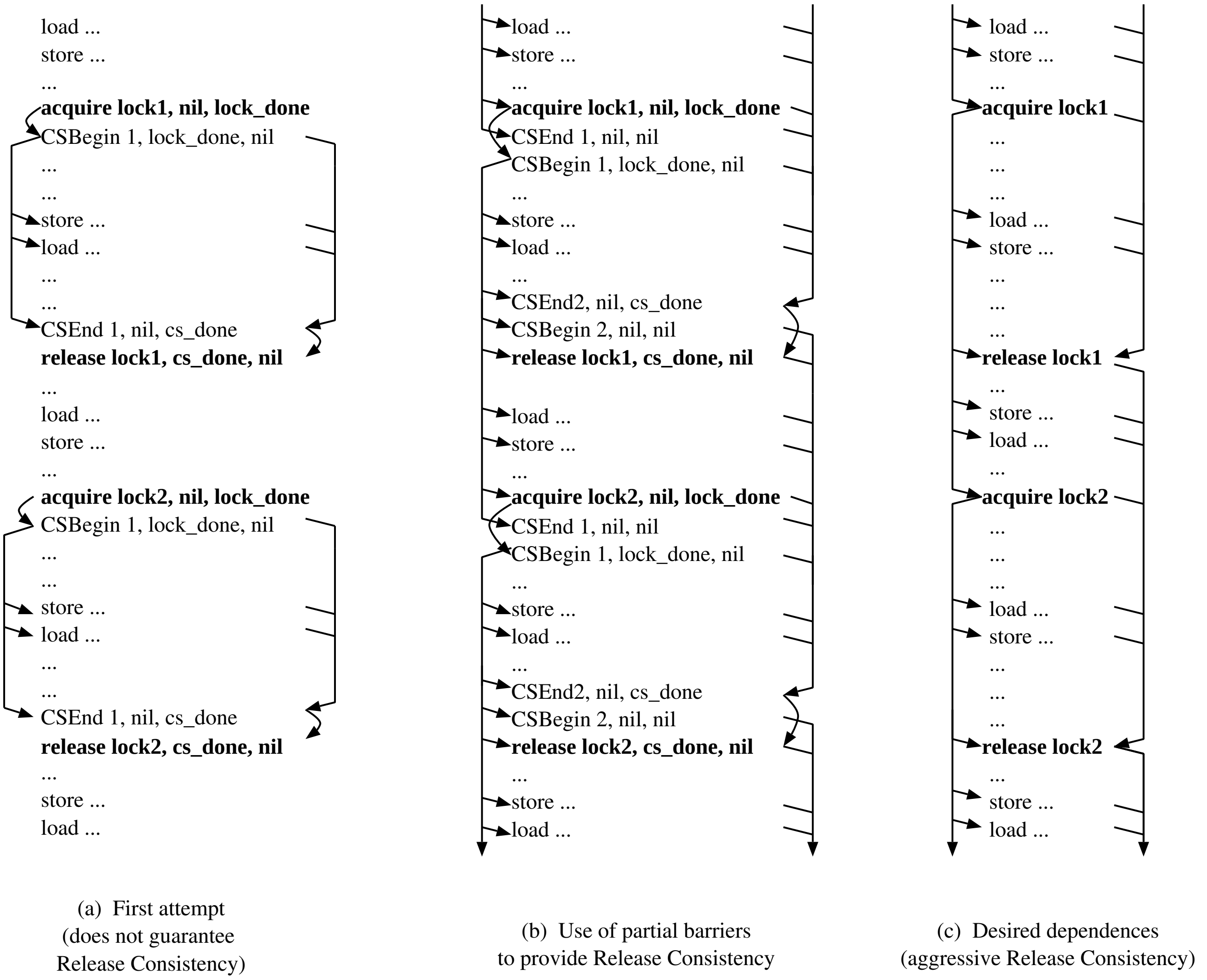

Figure 5: Use of $CSBegin$ and $CSEnd$ to provide RC

new instructions and special cache hardware to allow greater instruction-level parallelism, while still providing the proper ordering of accesses with respect to the synchronization operations.

First, we implemented a small (16-entry), fully-associative "lock cache" contained within the L1 cache, modeled after the synchronization buffer in [20]. Each entry in the lock cache holds a line for cache data. We add two new instructions, $acquire\ r_{addr}$, $r_{SynDepIn}$, $r_{SynDepOut}$ and $release\ r_{addr}$, $r_{SynDepIn}$, $r_{SynDepOut}$. When an acquire instruction is executed, the lock cache handles the request as in the synchronization buffer — the hardware retries the acquire until it succeeds, at which point the acquire instruction is allowed to complete. Notice that our simulator supports split-phase transactions, so the processor and cache can continue generating and handling other requests and replies while the acquire is being retried.

One interesting difference between our lock cache and the synchronization buffer in [20] is, since the status of a lock is determined by whether or not it is held in a lock cache, no status bits or words within the data line (or anywhere in the programmer's address space) need to be used to hold this state, thus the entire cache line can be used to hold data. This allows the programmer to take even better advantage of collocation of data, and makes the acquire an explicit prefetch operation. It also trivially supports dynamic binding of locks to objects.

4.4 Lock cache queues

For each lock, the QOLB [14] system keeps a linked list of nodes that are waiting for this lock. This allows the lock to be forwarded directly to the next node, rather than allowing all of the nodes to attempt an acquire. We similarly augment our lock cache with additional storage for such a list, although for implementation reasons our list is kept completely at the lock holder and is forwarded with the lock when the lock is released. Provided the number of nodes in a system is small, the extra storage to hold this information in a non-distributed manner is negligible: our simulator provides an extra cache line (32 bytes) for each line in the lock cache to hold this array, so that each lock can support concurrent acquire attempts from 32 other nodes.

This queuing mechanism provides three benefits. The first is a reduction in bus traffic compared to traditional `Test&Test&Set` acquires, similar to that achieved via QOLB [1]: the traffic for a k-contended lock (k processors are attempting to acquire it) drops from $O(k^2)$ to $O(k)$ bus messages in an invalidation-based protocol. The last two points are that the lock no longer suffers from either starvation or unfairness, due to the FIFO mechanism.

5 Simulation Methodology

5.1 Simulator

We used the Armadillo multiprocessor simulator, which was configured for a shared-bus snooping, invalidation-based shared memory system with a small number of nodes. The simulator models an aggressive PowerPC-instruction-set superscalar processor, with features including n-way superscalar execution, branch prediction and speculative execution, load forwarding, split-transaction buses (including multiple outstanding transactions by each bus member), and out-of-order retirement. For accuracy reasons, Armadillo uses program-driven simulation, that is, each instruction's effects are modeled, including register renaming, scheduling, *etc.* (contrasted with trace-based simulation, where typically only a subset of the instruction stream is modeled in detail, and direct-execution simulation, where "uninteresting" instructions are executed directly, and only "interesting" instructions are simulated). We assume a perfect instruction cache, and simulate both private and shared memory accesses to a 64K L1 cache.

Some of the configuration parameters for the simulations are listed in Table 1. We intentionally do not model some aspects of processor resource contention in order to examine theoretical limits to ILP, rather than the limits of a particular configuration. A configuration with a fast shared bus corresponds to a multiple-processors-on-a-chip [19] system, while a slow bus would represent a more traditional SMP system. Note that the memory latency for a cache fill is 14 cycles, as follows: 1 bus cycle to send the request, 9 bus cycles of latency (during which the bus is free for other requests), followed by 4 bus cycles of data.

Number of processors	4,8,16
Functional units: Int/FP/Mem	4/4/2 (2 ports to L1)
Write buffer	infinite, with load forwarding
Max outstanding L1 accesses	32
Instruction window size	varied, up to 256
Latencies	roughly same as PowerPC 604
Spec. Execution	3 pending branches
Branch Prediction	Two-level adaptive, 4K entries, 4 bits history, infinite RAS stack
L1 cache	64K 4-way, 32-byte lines, 1 cycle access
Bus	64-bit, varied slowdowns of 3:1 and 10:1, supports split transactions
Memory latency	14 bus cycles for 32-byte fill (1 for request, 9 for latency, 4 for burst fill)

Table 1: Processor Configurations

Several of the parameters were varied, to examine their effects upon performance improvements: the number of processors, the speed of the shared bus, and the size of the instruction window.

5.2 Benchmarks

We used several benchmarks from the SPLASH-2 suite [22] as examples of coarse-grain parallelism. We also modified a QuickSort benchmark from [16] and wrote a microbenchmark, `hist`, as examples of fine-grain parallelism. The problem sizes were reduced significantly to shorten the simulation time, which allowed us to study a wider range of configuration parameters. Several larger simulations were performed, to verify that the shorter simulations were still representative.

We augmented the simulator to provide detailed cycle-by-cycle profiling output. Notice that the profiling did not require any instrumentation of the benchmarks, so the benchmark execution time was unaffected. On a superscalar processor, determining the amount of time spent executing a particular function is difficult, since the executions of functions overlap in general. We chose the following metric: in each cycle, the oldest instruction in the instruction window is charged one tick. In this way, functions whose execution is completely overlapped with a previous function are charged 0 ticks, and the total number of ticks for all functions will be exactly the execution time. Notice that this method of profiling exposes the critical path instructions/functions.

The benchmarks, their problem sizes, and a few of their characteristics are presented in Table 2. Note that only the parallel portion of the benchmarks are detailed here. Any configuration-specific numbers (such as instruction counts and running time percentages) are from the base configuration of 4 nodes, 3:1 bus, an instruction window of size 32, and lock operations implemented in the traditional manner of `Test&Test&Set` with all-to-all memory barriers.

Profiling of initial experiments using the `qs` benchmark showed that up to 20% or more of the execution time was spent waiting for the first large task to be partitioned into P pieces. The benchmark was modified to perform a few (parallelized) partitioning steps first, until there was one task on the task queue for each processor, and then switch to the task-queue-based sorting. The parallel partitioning time is included in the measurements.

The `hist` benchmark is a simple histogram code. Each processor performs a histogram on its share of 1024 random integers between 1 and m, but rather than performing local histograms and then

Name	Parameters	# Insns.	Running Time in cycles (Base)	# Lock Acquires (Base)	Percent spent in lock/unlock (Base)	Avg. Cycles spent locking and unlocking (Base)
`barnes`	64 bodies, 4 steps	2.05M	550K	288	3.81%/0.13%	299.9
`ocean`	10x10, 5 steps	7.26M	1.77M	1616	4.18%/0.54%	206.5
`water-n`^{2}	27 molecules, 3 iters	6.93M	1.33M	388	0.25%/0.03%	38.7
`qs`	16384 ints	5.10M	926K	273	0.33%/0.06%	50.8
`hist`	1024 ints, 1 bin	279K	133K	1024	60.1%/0.58%	315.3
`hist`	1024 ints, 256 bins	58K	33K	1024	59.7%/2.39%	79.4

Table 2: Benchmark Characteristics

merging these histograms together, we use only a single global histogram array, in order to force a high degree of contention. In addition, by varying the parameter m we can control the degree of contention: with $m = 1$, each processor is attempting to update the same shared counter, while with $m = 256$ it becomes less likely that contention will occur.

5.3 PARMACS Macros

The SPLASH-2 benchmarks [22] are written to use the PARMACS macros [6]. Our techniques can be completely hidden within library routines, so we were able to write six different PARMACS files and compile the benchmarks for the different techniques without source code modification. The SPLASH-2 benchmarks use two lock primitives, `LOCK(lock)` and `UNLOCK(lock)`, which are replaced by calls to functions specified in the PARMACS macro file. In the PARMACS file for our base configuration, `LOCK()` uses a `Test&Test&Set` loop (implemented with `compare-and-swap`) followed by an all-to-all memory barrier. `UNLOCK()` uses an all-to-all memory barrier followed by a store. The remaining PARMACS files can be described via what features they add with respect to the base:

+Acq: `LOCK()` uses an `Acquire` instruction, followed by an all-to-all memory barrier. `UNLOCK()` uses an all-to-all barrier followed by a `Release` instruction. Relative to the base configuration, `+Acq` removes the conditional branches from the acquire operation. The lock cache is used, but acquires are retried on the bus until they are successful.

+Q: The queuing feature of the lock cache is enabled. This reduces the bus traffic from $O(k^2)$ to $O(k)$. `+Q` requires `+Acq`.

+PB: The partial barriers were used. This allows more memory access reorderings to take place.

6 Analysis

Since most of the coarse-grained benchmarks performed a small number of lock operations, the overall speedup from these techniques is usually quite small (although some applications achieved speedups of 10% or more due to the small problem sizes). Instead, we focus on the absolute reduction in execution time in cycles. For example, `ocean` on 4 processors with a fast 3:1 bus and a 32-entry instruction window takes 1,770,626 cycles under the base configuration (`Test&Test&Set` with all-to-all memory barriers), and 1,652,909 using acquires and partial memory barriers, with lock cache queuing. This difference is 117,717 cycles, or roughly a 6% improvement. More importantly, there are 1,616 lock acquire/release pairs performed by all processors, and since only lock operations were changed, we have an average improvement of 72.8 cycles per lock over the base model.

Early simulation results showed that an instruction window size of 256 did not make very much difference in the performance — in fact, for some benchmarks, the larger instruction window increased the branch misprediction penalty without improving performance significantly, leading overall to a longer execution time. We present savings-per-lock-operation results for the 32-entry instruction window for 4, 8, and 16 processors in Figures 6 through 8, and for the 256-entry instruction window for 8 processors in Figure 9.

In general, providing blocking acquires (`+Acq`) provided the highest performance improvement, and `+Q` and `+PB` provide smaller benefits (and `+PB` by itself provides very little benefit). These results match our expectations: by removing the conditional branch (`+Acq`), we allow the processor to examine a much larger window of non-speculative instructions. Without `+Acq`, partial barriers are unable to extract much non-speculative memory-access parallelism, and thus provide little performance improvement by themselves.

Several data points show slowdowns when only `+PB` is in effect, such as `ocean` and `qs` under 16 processors. Investigation showed that under `+PB`, on average each lock acquire generated an extra fill

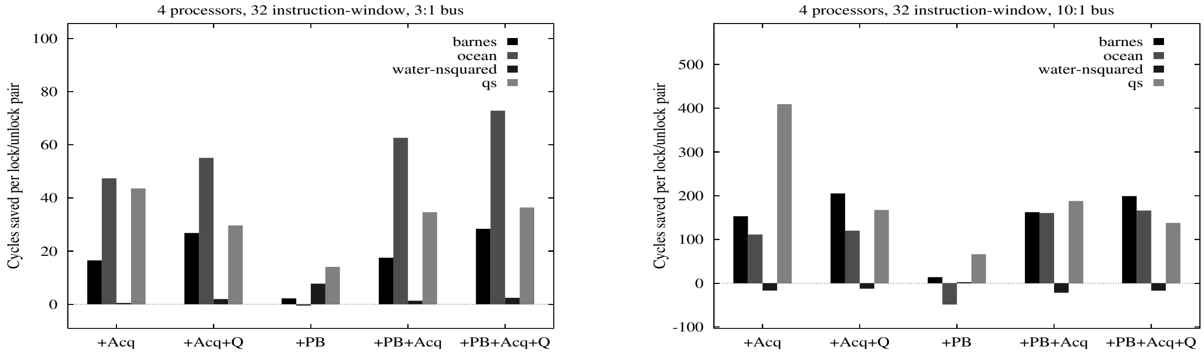

Figure 6: Four-processor results for fast and slow bus speeds

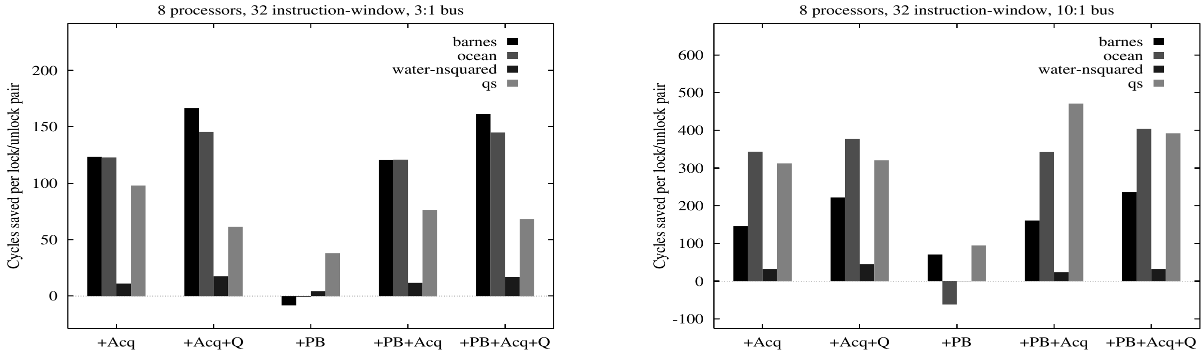

Figure 7: Eight-processor results for fast and slow bus speeds

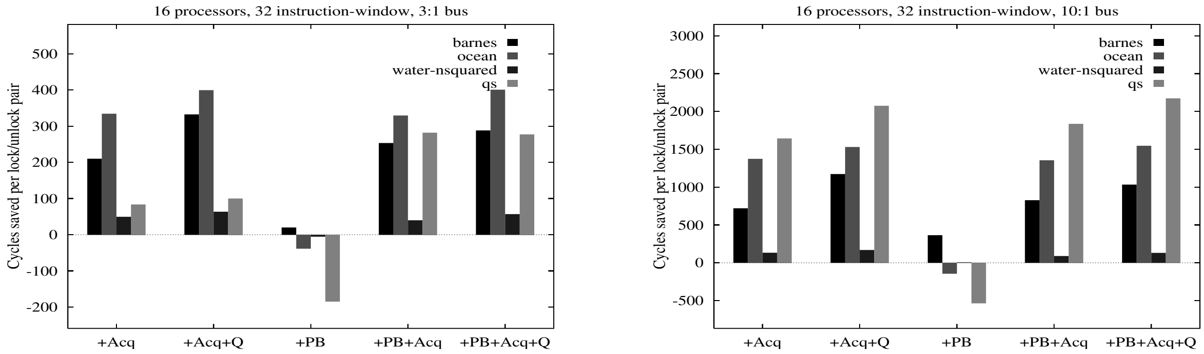

Figure 8: Sixteen-processor results for fast and slow bus speeds

request and an extra ownership request. This led to higher bus utilization, which slowed the release of the lock and thus the overall execution time. The situation for `qs` is even worse, since it is not deterministic: by delaying the lock release, the tasks were pushed and pulled from the centralized work queue in a slightly different order, and the load became more unbalanced, leading to a longer execution

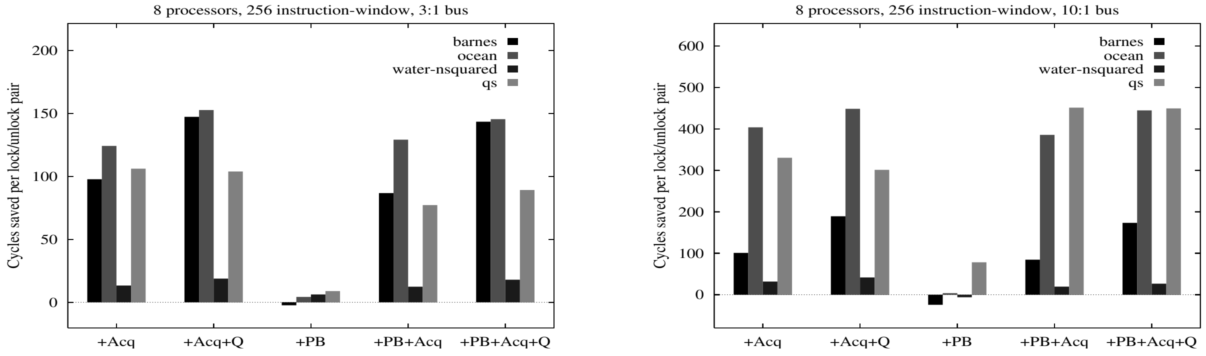

Figure 9: Eight-processor results for fast and slow bus speeds with a 256-entry instruction window

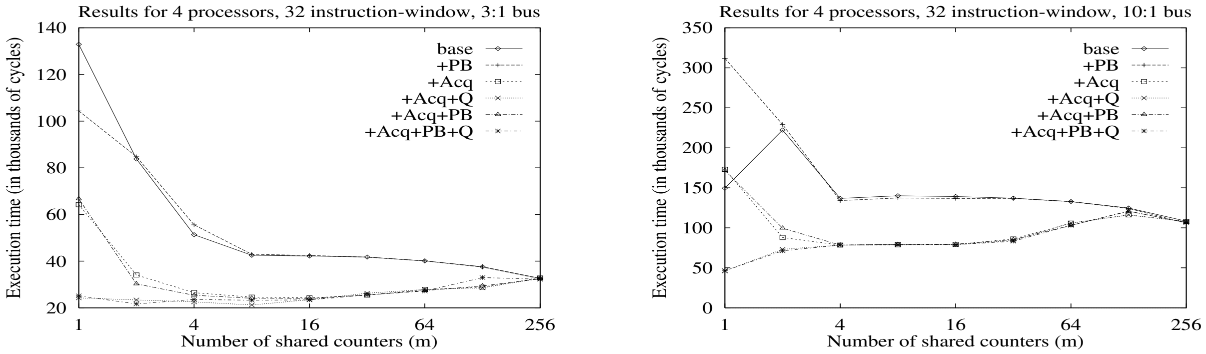

Figure 10: Four-processor `hist` results

time compared to the `Test&Test&Set` and all-to-all barrier timings.

For the histogram benchmark `hist`, the instruction window size makes very little impact on performance, so we only show an instruction window of size 32. The execution time is graphed against the number of shared counters in Figures 10 through 12. As expected, under high contention, blocking acquires with queuing provides substantial benefit. In fact, the execution time is relatively flat for `+Acq+Q` until the benchmark uses 32 or more counters, at which point the higher number of "cold" misses by the lock cache increases execution time. `+PB` provides little benefit, since there is very little memory access parallelism to exploit.

7 Conclusion

We have presented architectural features that together allow a highly aggressive implementation of Release Consistency for ILP processors with speculative execution. These same architectural features

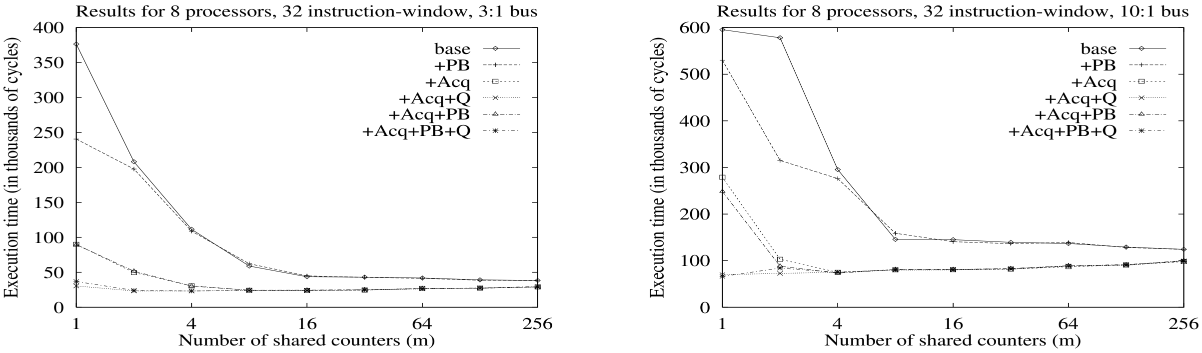

Figure 11: Eight-processor `hist` results

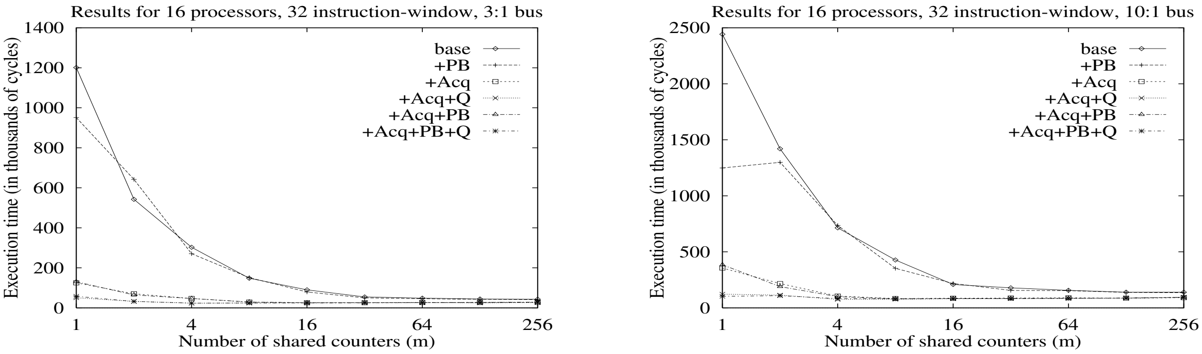

Figure 12: Sixteen-processor `hist` results

can also be used to provide even weaker memory consistency models, allowing a programmer to tailor the locking primitives to obtain higher performance for applications where Release Consistency is overly strong. The implementation of such an aggressive RC model can be completely hidden from the user in library functions, without modifications to either the compiler or the language.

There is still a substantial performance gap between typical RC implementations and a highly aggressive one, especially for fine-grain applications. Removing the control-flow dependences provided the largest performance improvement in general, but queuing and partial memory barriers provided significant additional savings. For fine-grain applications, the results were even more encouraging: in the single counter version of our `hist` microbenchmark with 8 or 16 processors, the speedup using these techniques was around 10 or more.

References

[1] Nagi M. Aboulenein, Stein Gjessing, James R. Goodman, and Philip J. Woest. Hardware Support for Synchronization in the Scalable Coherent Interface (SCI). Technical Report Computer Sciences Dept. Tech. Report 1117, University of Wisconsin-Madison, November 1992.

[2] Sarita Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical Report WRL Research Report 95/7, Digital Western Research Laboratory, 1995.

[3] Sarita V. Adve and Mark D. Hill. "Weak Ordering — A New Definition," In *Seventeenth International Symposium on Computer Architecture (ISCA 1990)*, pages 2–14, 1990.

[4] Sarita V. Adve and Mark D. Hill. "A Unified Formalization of Four Shared-Memory Models," *IEEE Transactions on Parallel and Distributed Systems*, 4(6), 613–624, June 1993.

[5] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, 1991.

[6] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.

[7] 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, 1986.

[8] Guang R. Gao and Vivek Sarkar. Location Consistency: Stepping Beyond the Barriers of Memory Coherence and Serializability. Technical Report ACAPS Technical Memo 78, McGill University, 1993. A condensed version appears in the Proceedings of the 1996 Conference on Parallel Processing.

[9] Kourosh Gharachorloo, Daniel 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, 1990.

[10] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report no. 61, SCI Committee, 1989 (also available as University of Wisconsin-Madison Technical Report CS-TR-91-1006).

[11] James R. Goodman, Mary K. Vernon, and Philip J. Woest. "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors," In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 64–75, April 1989.

[12] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, 12(3), 463–492, July 1990.

[13] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA-96)*, June 1996.

[14] Alain Kägi, Doug Burger, and James R. Goodman. "Efficient Synchronization: Let Them Eat QOLB," In *Twenty-fourth International Symposium on Computer Architecture (ISCA 1997)*, June 1997.

[15] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs," In *IEEE Transactions on Computers*, volume C-28(9), pages 241–248, 1979.

[16] Honghui Li, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. "Message Passing Versus Distributed Shared Memory on Networks of Workstations," In *Supercomputing '95*, 1995.

[17] David Mosberger. "Memory Consistency Models," *Operating Systems Review*, 27(1), 18–26, January 1993.

[18] Motorola Corporation. *PowerPC Microprocessor Family: The Programming Environments*, 1994.

[19] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. "The Case for a Single-Chip Multiprocessor," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[20] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 12–23, 1996.

[21] SPARC International, Inc. *The SPARC Architecture Manual, Version 9*, 1994.

[22] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," In *Twenty-second International Symposium on Computer Architecture (ISCA 1995)*, pages 24–36, June 1995.