Bringing Sanity to the Semantics of Memory Access Instructions in Multiprocessors

Computation Structures Group Memo 423
May 1999 (Revised: July 1999)

Xiaowei Shen, Arvind and Larry Rudolph
xwshen, arvind, rudolph@lcs.mit.edu

Bringing Sanity to the Semantics of Memory Access Instructions in Multiprocessors

Xiaowei Shen, Arvind, Larry Rudolph
Laboratory for Computer Science
Massachusetts Institute of Technology
xwshen, arvind, rudolph@lcs.mit.edu

Abstract

Most weaker memory models have arisen as a consequence of some architectural optimization in the implementation of memory access instructions, rather than from some grand high-level design. Many of these optimizations are transparent for uniprocessors but can be observed in multiprocessor because they break the Sequential Consistency model in one way or another. As a partial remedy all modern microprocessors now provide memory fences to control instruction reordering and overload their semantics to imply something vague about store completion.

This paper shows that CRF, our recently proposed mechanism-oriented memory model that exposes both instruction reordering and data replication, can be used as the universal interface between the compiler writer and the underlying architecture. Programs written under sequential consistency and various weaker memory models can be translated into CRF programs without incurring unnecessary overhead. Also, CRF programs can be mapped back to programs that can run efficiently on existing microprocessors. In particular, we show translations between CRF and IA-64, Intel's newly released instruction set architecture. The translations show some of the weaknesses of Intel's specification in regards to store atomicity.

1 Introduction

Everchanging memory models and their complicated, imprecise definitions have caused a reaction to go back to the simple, easy-to-understand sequential consistency (SC) memory model [9], even though there are a plethora of problems in its high-performance implementation and no compiler writer seems to adhere to its semantics. The newly released IA-64 instruction set [12] of Intel presents yet another new, imprecise memory model, and just adds to the pressure to correct the situation. Recently, we have proposed a new mechanism-oriented memory model called Commit-Reconcile & Fences (CRF), which exposes both data replication and instruction reordering at the ISA level [16]. The motivation underlying CRF is to eliminate the *modèle de l'année* aspect of existing weaker memory models and, simultaneously, allow high-performance implementations.

There are good reasons to be skeptical of yet another memory model. For most computer manufacturers a new memory model would be a non-starter, if it were incompatible with the memory model embedded in their hardware or software products. It is essential to have upward compatibility, i.e., the ability to run existing programs correctly on a machine with the CRF model. Furthermore, if compiler writers adopt CRF as the target model then it is important to have downward compatibility, i.e., the ability to run CRF programs well on existing machines. Indeed, CRF has both of these properties vis-a-vis most other existing memory models.

We will show that programs written under various memory models, such as, SC, IBM 370, Sparc's TSO, PSO and RMO models [18], and Intel's IA-64 [12] can be translated into CRF programs such that the translated programs show exactly those behaviors that are permitted by the original models. These translations can be taken as a precise definition of the often imprecise description of these memory models by the manufactures [1]. We will present algorithms that translate a sequence of memory access instructions generated by a processor with a particular memory model into a sequence of CRF instructions. The translations are simple enough to be directly implementable in hardware. The translation schemes should convince the reader that the CRF model provides an upward compatible way for building the next generation of microprocessors.

The CRF model can provide immediate relief to compiler writers by isolating them from the plethora of memory models. We will show that it is possible to statically translate CRF programs to exploit the memory models of existing microprocessors to the fullest extent. These translation schemes simply replace CRF instructions by ordinary load, store and memory fences, if they are available. Even in the absence of CRF microprocessors, CRF can serve the purpose of an implementation-independent intermediate representation for various underlying architectures that support different memory models. We will also discuss translations of high-level programming models, such as data-race-free programs, into CRF programs.

Our model covers the spectrum from uniprocessor to SMP to DSM. CRF semantics permit most, if not all, current architectural optimizations. The semantics are clear and precise so that there are no fuzzy cases. Moreover, its semantics cover all types of programs, not only those that satisfy certain synchronization criteria. So, it will be clear what a machine will do when programs with data races become fashionable.

A word of caution is in order. Memory models are notoriously difficult to define precisely. We have followed the usual practice of defining a memory model as the relation between a set of legal output sequences in response to a set of input sequences of memory instructions. Such a relation can be specified precisely, but one also needs to address, given a program, how a memory access instruction stream is generated. Architectural mechanisms such as register renaming and speculative execution, by issuing memory access instructions out-of-order, can affect the program behavior that can be observed in a multiprocessor setting. To date such properties of instruction issue have not been specified precisely. However, it is often possible to reason about program behaviors by making conservative assumptions (e.g., no speculation) about instruction issue.

We begin by showing how the differences in implementations of memory access instruction which are not visible in uniprocessors become visible in multiprocessors (Section 2). Section 3, after reviewing the definition of the CRF memory model, describes some of its properties which are useful in understanding CRF's universality. Section 4.1 gives translation schemes for programs written under various memory models into CRF, while Section 4.2 explains how to run CRF programs on existing microprocessors. Finally, Section 5 presents our conclusions.

2 Why Memory Instruction Semantics Matter

Many people have difficulty understanding why and how a memory model can affect them. Does it have impact on the way they write programs? Does it improve the performance? Does it affect the compiler or microarchitecture optimizations? The simple answer is that if we stick to uniprocessor systems and sequential programming the memory model is irrelevant. But, we rarely do! Even sequential programmers may initiate input/output tasks that run concurrently with the main program, and uniprocessors may have operating systems that are expressed as "cooperating sequential tasks." The fact is that the memory model affects everything but often in insidious and

[1]This is a classic case of the "Chicken and Egg" problem - if the model is not defined precisely in the first place, how can we be sure that its CRF definition is correct? Oh well.

surprising ways.

Many architectural mechanisms that are transparent in uniprocessors suddenly become visible in multiprocessors. A difference in implementation of memory related instructions can cause a subtle difference in observable behavior, giving rise to a different memory model. That is, the range of permissible behaviors is affected by just about every uniprocessor optimization. The fact that these range of behaviors have a lot of commonality or that the simple sequential execution of instructions is always a permitted behavior is hardly a source of comfort, because it is the additional behaviors that make parallel programs go wrong in surprising ways. Therefore, there is a need to pin down the exact definitions of memory instructions for multiprocessors. Since the same microprocessors and operating systems are used in uniprocessors and multiprocessors, a microprocessor designer has no option but to pay close attention to these subtle differences.

In this section we will show via examples the subtle differences that arise in program behaviors because of different implementations of load and store instructions. Some of these differences are observable even in a uniprocessor system when concurrent programs are executed. Many examples in this section are known to experts in memory models (see, for example, the excellent tutorial by Adve and Gharachorloo [1]) but are presented here from a different perspective. *Note that in all the examples in this paper, the initial value of a variable is assumed to be zero.*

2.1 Non-transparency of Uniprocessor Optimizations

Although many optimizations are transparent with sequential programming on uniprocessor systems, they can often be exposed programmatically on multiprocessor systems. This subsection considers instruction reordering and shows that a programmer must know all of the implementation details in order to understand the semantics of the program.

Unlike a compiler, an architecture almost never reorders instructions unless it has to. For example, a pipelined architecture reorders instructions dynamically only when an instruction gets stalled waiting for input data or a resource, and the following instruction is ready to proceed. A more subtle reordering occurs because of store instructions. When should a store instruction be considered to have completed? When the value has been deposited in some store buffer or cache? Or only when the main memory has been updated? In fact, the instructions are never reordered before decoding, but the effect of out-of-order instruction dispatch or completion can often be explained as if the instructions had been reordered statically.

Example 1 (Store buffers): One of the earliest architectural optimizations was to let a load instruction bypass the store instructions which were waiting in the store buffers, as long as there was no address conflict between the load and store addresses (e.g. IBM 370). An architecture that allows loads to overtake stores must provide some mechanism (e.g. memory barrier instructions) to enforce the ordering between a store and a following load whenever necessary.

Example 2 (Short-circuiting): It is common for an architecture with store buffers to allow "short-circuiting," that is, a load is allowed to obtain the data from the store buffer if the address is present there. Sparc models allow this while IBM 370 does not. This feature can be exposed in a multiprocessor; the extra load on each processor would behave as a memory barrier on IBM 370, preventing the following load to be performed before the preceding store completes. The extra load will make no difference on Sparc with TSO, because short-circuiting would allow the extra load to complete.

Proc 1	*Proc 2*	
Store(flag_1,1); r_1 := Load(flag_2);	Store(flag_2,1); r_2 := Load(flag_1);	**Example 1:** If a load may overtake a store r_1=0 and r_2=0 is possible
Store(flag_1,1); r_3 := Load(flag_1); r_1 := Load(flag_2);	Store(flag_2,1); r_4 := Load(flag_2); r_2 := Load(flag_1);	**Example 2:** On Sparc, extra load has no effect but acts as barrier on IBM 370.

Store(x,1);
Store(flag,1);
	r_1 := Load(flag);
	r_2 := Load(x);
		Example 3: With non-FIFO store buffers r_1=1 and r_2=0 is possible

Store(x,1);
Store(flag,1);
	r_1 := Load(flag);
	r_2 := Load(x);
		Example 4: Even when stores are ordered, it is possible for r_1=1, r_2=0.

Store(flag_1,r_1); r_1 := Load(flag_2);	Store(flag_2,r_2); r_2 := Load(flag_1);	**Example 5:** Register renaming removes dependencies, so r_1=0 and r_2=0.

Store(x,1);
Store(flag,1);
	L: r_1 := Load(flag);
	Jz(r_1,L);
	r_2 := Load(x);
		Example 6: Speculative Load(x) may allow r_2=0 and r_1=1 even if the stores are ordered.

Figure 1: Examples for Instruction Reordering

Example 3 (Non-FIFO store buffers): Some architectures, such as IBM 370 and Sparc's TSO model, assume that the data is moved from a store buffer to a cache or the main memory in a strictly FIFO manner. Some architectures, such as Sparc's PSO model, do not follow this restriction.

Example 4 (Non-blocking caches): Modern architectures such as Sparc V9 [18], Alpha [17] and PowerPC [14], have non-blocking caches and allow multiple outstanding loads. This effectively allows even more relaxed memory models because both stores and loads can bypass outstanding loads. Thus, two ordered stores may be observed to happen in the reverse order if loads can be reordered.

2.2 Register Renaming and Speculative Execution

A memory model defines the interface between the programmer and the underlying memory system. Given a sequence of memory access instructions on each processor, the memory model specifies a set of legal values that can be returned for each load access throughout the program execution. However, such a definition of memory models cannot fully define the program behaviors because of the lack of specification about how memory access streams are generated from a given program. Speculative implementations can have subtle impact on program behaviors. Even register renaming, which has nothing to do with memory access instructions directly, can ~~also~~ affect the program behavior.

Example 5 (Register renaming): Initially registers r_1 and r_2 contain zeros. Without register renaming, the anti-dependency on each processor behaves as an implicit memory barrier between the store and the following load. Thus register renaming will allow reordering of loads and stores and introduce new behaviors.

Example 6 (Speculative loads): The question is whether a branch instruction behaves as an implicit barrier. With speculative execution, a processor may speculatively perform a load of one

Proc 1	*Proc 2*	*Proc 3*	*Proc 4*	
Store(x,1);	Store(x,2);	r_1 := Load(x); r_2 := Load(x);	r_3 := Load(x); r_4 := Load(x);	**Example 7:** Can r_1=1,r_2=2 but r_3=2, r_4=1?
Store($flag_1$,1);	r_1 := Load($flag_1$); Store($flag_2$,1);	r_2 := Load($flag_2$); r_3 := Load($flag_1$);		**Example 8:** if r_1=1,r_2=1 can r_3=0?

Figure 2: Examples for Store Atomicity

location before it observes the signal value of another location.

Value speculation has been an active research area in recent years because of its potential for performance improvement. All speculative execution mechanisms and compiler optimizations can be considered as special cases of value speculation. Unfortunately, the impact of value speculations on memory models, has not been studied. From these simple examples it should be clear that value speculation will make it very difficult to characterize the range of acceptable behaviors for memory related instructions. [2]

2.3 Multiprocessor Concerns

Memory access operations in Distributed Shared Memory (DSM) multiprocessors can take a long time to complete, often involving numerous communications between nodes. In a system without clear semantics, the actual implementation can often affect a program's behavior.

If caches or, in general, data replication is to remain transparent in a multiprocessor, each store operation must be atomic. Informally, a store is atomic if the effect of the store becomes visible to all the processors in a lock step. The following examples show the effect of non-atomic stores.

Example 7 (Non-atomic stores on a single location): Consider the situation in which two processors modify a location x while two other processors read it. Assuming that memory accesses are executed in order on each processor, an interesting question is whether the two stores can be observed in different orders by different processors. In other words, in Figure 2, is it possible for r_1 and r_2 to get 1 and 2, respectively, while r_3 and r_4 to get 2 and 1, respectively? Obviously this can happen in the presence of non-atomic stores.

We use the term *store atomicity* [4, 5] to mean that such behavior is impossible.

Example 8 (Causality violation): Problems with non-atomic stores also occur in programs with multiple locations. For example, processor 1 asserts $flag_1$; processor 2 reads $flag_1$ and then sets $flag_2$; and processor 3 reads $flag_2$ and then $flag_1$. Assume each processor executes memory accesses strictly in order.

When one processor (processor 2) observes the effect of a store, can it deduce that another processor (processor 3) will be able to observe the same effect? Such causality can be violated easily when stores are performed non-atomically.

Enforcing atomicity in multiprocessors requires extra support in the form of cache coherence protocols. The most common method to enforce store atomicity is to allow a processor to write to

[2]The IA-64 architecture provides new instructions to allow aggressive speculative executions. However, the IA-64 definitions only address the issue of raising exceptions and have little to say about program behavior in multiprocessor systems.

its cache only after the address has been invalidated in all other caches. This can be an expensive operation in large DSM systems, in terms of both the latency and the number of messages that need to be issued. While store atomicity may be natural to most invalidate-based protocols, ensuring such atomicity can be very expensive for update-based protocols in DSM systems. This is because, when a processor performs a store, it must first invalidate all other caches before it can start updating them to ensure that the new value is observable in a lock step.

A memory model that allows non-atomic stores can allow more design flexibility for cache coherence protocols. Consider a simple update-based protocol that makes no effort to ensure store atomicity. In the protocol, when a processor performs a store, it writes the data to its cache and multicasts the data to all the other caches where the address is also cached. When a cache receives the data, it updates its copy and sends an acknowledgment to the processor where the store is performed. The store is considered completed when all the acknowledgments regarding the update data have been received. The reader can verify such a protocol can easily cause stores to be performed in some non-atomic fashion in DSM systems.

It is worth emphasizing that we do not see the utility of such semantics. That is, we want split-phase memory access operations, but not every solution is acceptable. The description of PowerPC [14] seems to allow non-atomic stores, and the description of the IA-64 is not clear on this issue.

Release consistency [7] and its variations [13, 3] also do not rely on atomic stores. They improve performance by exploiting the properties of *data-race-free* programs [2, 6]. In a *data-race-free* program, a thread can modify a shared location only after acquiring a lock. An implementation can postpone making the effect of stores visible to other threads (processors) until the lock is released or better yet, until the lock is acquired by another thread [13]. The effect of a store in such models has to be made visible only to that thread that is successful in acquiring the lock. The idea has not found much favor with architects because instruction sets do not permit one to recognize Acquires and Releases of a lock, or the locations being guarded by it. The IA-64 of Intel does provide some relief in this direction.

2.4 How to deal with this complexity

The examples in this section taken collectively illustrate that just about every innovation in processor architecture has affected the memory model. The everchanging and implementation-dependent memory models definitions have created a situation where even experienced architects have difficulty articulating precisely the impact of these memory models on program behaviors. It is not uncommon that memory model specifications in modern microprocessor manuals are ambiguous or even wrong. Given such a state of affairs, it is not surprising that not only compiler writers and systems programmers, but also architects want a simpler and cleaner memory model. Since SMP's and other servers represent the most profitable and one of the fastest growing sectors of computer hardware business, this problem needs an urgent solution.

There are two possible solutions. One is to adopt SC as the memory model. The main argument is that aggressive speculative executions can allow SC to be implemented in a much more efficient way than one could imagine even a few years ago [19, 11, 8]. Speculative implementation of SC, however, has some serious potential problems that have not been investigated yet. For example, it is not clear whether such implementations are scalable. The issue becomes far more serious for DSM systems, in which memory access latencies are often large and unpredictable. Another issue is that speculative implementations of SC may exacerbate the false sharing problem, which is already quite serious in SMP's. Compiler writers like SC because of its stability – not necessarily for its semantic properties that limit their ability to exploit the high performance of the underlying

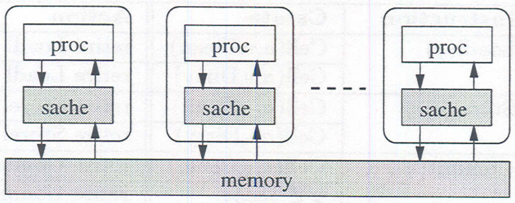

Figure 3: Semantic Configuration of CRF

microarchitecture.

The second solution is to semantically expose the split-phase nature of memory access operations. One way to do this is to expose data replication. That is, a store initially is placed in a local cache and some time later it is copied to a globally visible memory. Load operations fetch from the local cache and data is copied from the globally visible memory to the local cache. The following section makes these semantics precise, and leave no room for ambiguity in the interpretation of the examples presented above. The semantics of our proposed CRF memory model allows most high-performance implementation techniques used in contemporary microprocessors.

3 The CRF Memory Model

The CRF model exposes data replication via a notion of semantic cache (sache). Each processor is associated with a sache, on which memory access instructions are performed (see Figure 3). Each sache cell has an associated state, which can be either Clean or Dirty. The Clean state indicates that the data has not been modified since it was cached or last written back. The Dirty state indicates that the data has been modified and has not been written back to the memory since then. The description of CRF in subsections 3.1 and 3.2 is taken almost verbatim from another publication [16].

In CRF, conventional load and store operations are decomposed into finer-grain instructions. There are five memory-related instructions, Loadl (load-local), Storel (store-local), Commit, Reconcile and Fence. The CRF model can be specified by two sets of rules, the CR rules and the reordering rules.

3.1 The CR Rules

Figure 4 presents the rewriting rules that specify the CRF model (except reordering rules). It contains one rule for each instruction and three background rules that operate between saches and memory. A Loadl instruction reads the data from the sache if the address is cached; otherwise it stalls until the value of the address is somehow fetched into the sache. A Storel instruction writes the data into the sache if the address is cached, and sets the sache state to Dirty. Since CRF inherently allows a store operation to be performed without coordinating with other saches, different saches can have a cell with the same address but different values.

A Commit instruction on a dirty cell stalls until the data is written back to the memory, while a Reconcile instruction on a clean cell stalls until the data is purged from the sache. The Commit and Reconcile instructions can be used to ensure that the data produced by one processor can be observed by another processor whenever necessary. The memory behaves as the rendezvous between the writer and the reader: the writer performs a commit operation to guarantee that the

Processor Rules				
Rule Name	**Instruction**	**Cstate**	**Action**	**Next Cstate**
CRF-Loadl	Loadl(a)	Cell(a,v,Clean)	retire Loadl	Cell(a,v,Clean)
		Cell(a,v,Dirty)	retire Loadl	Cell(a,v,Dirty)
CRF-Storel	Storel(a,v)	Cell(a,-,Clean)	retire Storel	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	retire Storel	Cell(a,v,Dirty)
CRF-Commit	Commit(a)	Cell(a,v,Clean)	retire Commit	Cell(a,v,Clean)
		$a \notin sache$	retire Commit	$a \notin sache$
CRF-Reconcile	Reconcile(a)	Cell(a,v,Dirty)	retire Reconcile	Cell(a,v,Dirty)
		$a \notin sache$	retire Reconcile	$a \notin sache$

Background Rules				
Rule Name	**Cstate**	**Mstate**	**Next Cstate**	**Next Mstate**
CRF-Cache	$a \notin sache$	Cell(a,v)	Cell(a,v,Clean)	Cell(a,v)
CRF-Writeback	Cell(a,v,Dirty)	Cell(a,-)	Cell(a,v,Clean)	Cell(a,v)
CRF-Purge	Cell(a,-,Clean)	Cell(a,v)	$a \notin sache$	Cell(a,v)

Figure 4: CRF Rules (except reordering rules)

modified data has been written back to the memory, while the reader performs a reconcile operation to guarantee that the stale copy, if any, has been purged from the sache so that the subsequent load operation must retrieve the data from the memory. Exploitation of the flexibility offered by the commit and reconcile operations underlies all the protocols presented in this paper.

While both Loadl and Storel instructions are local operations, data can be propagated between the memory and a sache under proper conditions. A sache can obtain a clean copy from the memory, if the address is not cached (thus it cannot contain more than one copy for the same address). A dirty copy can be written back to the memory, after which the sache state becomes clean. A clean copy can be purged from the sache at any time, but cannot be written back to the memory. The cache, writeback and purge operations are often referred as background operations since they can be invoked voluntarily even though no instruction is executed by any processor.

3.2 The Reordering Rules

CRF allows memory accesses to be reordered as long as data dependence constraints are preserved, and provides memory fences to enforce ordering if needed. Two memory accesses have data dependence if they both access the same address, and at least one of them is a Storel. There are four types of memory fences: Fence$_{\mathrm{rr}}$ (read-read), Fence$_{\mathrm{rw}}$ (read-write), Fence$_{\mathrm{wr}}$ (write-read) and Fence$_{\mathrm{ww}}$ (write-write). Each memory fence has a pair of arguments, a pre-address and a post-address, and imposes an ordering constraint between memory operations involving the pre- and post- addresses. For example, Fence$_{\mathrm{rw}}$(a_1,a_2) ensures that any preceding Loadl to location a_1 must be performed before any following Storel to location a_2 can be performed. This implies that instructions Loadl(a_1) and Storel(a_2,v) separated by Fence$_{\mathrm{rw}}$(a_1,a_2) cannot be reordered.

A Fence$_{\mathrm{wr}}$ or Fence$_{\mathrm{ww}}$ imposes ordering constraints on preceding Commit (instead of Storel) operations, since only a Commit can force the data of a Storel to be written back to the memory. It makes little sense to ensure that a Storel operation has been completed if it is not followed by a Commit. Similarly, a Fence$_{\mathrm{rr}}$ or Fence$_{\mathrm{wr}}$ imposes ordering constraints on following Reconcile (instead of Loadl) operations, since only a Reconcile can force the stale data, if any, to be purged. It makes little sense to postpone a Loadl operation if it is not preceded by a Reconcile. Memory fences can always be reordered with respect to each other.

Figure 5 concisely defines the conditions under which two adjacent memory instructions can

$I_2 \Rightarrow$ $I_1 \Downarrow$	Loadl (a')	Storel (a',v')	Fence$_{rr}$ (a'_1,a'_2)	Fence$_{rw}$ (a'_1,a'_2)	Fence$_{wr}$ (a'_1,a'_2)	Fence$_{ww}$ (a'_1,a'_2)	Commit (a')	Reconcile (a')
Loadl(a)	true	$a \neq a'$	$a \neq a'_1$	$a \neq a'_1$	true	true	true	true
Storel(a,v)	$a \neq a'$	$a \neq a'$	true	true	true	true	$a \neq a'$	true
Fence$_{rr}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{rw}(a_1,a_2)$	true	$a_2 \neq a'$	true	true	true	true	true	true
Fence$_{wr}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{ww}(a_1,a_2)$	true	$a_2 \neq a'$	true	true	true	true	true	true
Commit(a)	true	true	true	true	$a \neq a'_1$	$a \neq a'_1$	true	true
Reconcile(a)	$a \neq a'$	true	true	true	true	true	true	true

Figure 5: CRF Reordering Table

be reordered (assume instruction I_1 precedes instruction I_2, and a 'true' condition indicates that the reordering is allowed)[3]. The underlying rational is to allow maximum reordering flexibility for out-of-order execution. For example, the rule represented by the Storel-Storel entry specifies that two Storel operations can be reordered if they access different addresses.

3.3 Coarse-grain CRF instructions and compiler optimizations

In CRF, it is always safe to insert commit, reconcile and fences without affecting the correctness of the program. That is, the extra commits, reconciles and fences can only decrease the number of behaviors shown by a program. This fact allows us to introduce coarse-grain versions of such instructions, which may be more practical at the instruction set level. For example, a coarse-grain fence imposes an ordering constraint with respect to address ranges instead of individual locations. An address range can be a cache line, a page or even the whole address space (we represent the whole address space by '*').

Note a compiler can reorder instructions according to the reordering rules of CRF. The compiler may increase the distance between a Loadl and its preceding Reconcile on the same address so that a prefetch can be used to hide the read latency. Likewise, the compiler may increase the distance between a Storel and its following Commit on the same address to hide the writeback latency.

A compiler can also eliminate unnecessary commit, reconcile and fences. For example, two back-to-back commit or reconcile operations on the same address can be merged into one. The following rules are some examples of compiler optimizations.

$$\begin{array}{lcl}
\text{Commit}(a);\ \text{Commit}(a) & \rightarrow & \text{Commit}(a) \\
\text{Reconcile}(a);\ \text{Reconcile}(a) & \rightarrow & \text{Reconcile}(a) \\
\text{Storel}(a,v);\ \text{Reconcile}(a) & \rightarrow & \text{Storel}(a,v) \\
\text{Storel}(a,v_1);\ \text{Storel}(a,v_2) & \rightarrow & \text{Storel}(a,v_2) \\
r_1 := \text{Loadl}(a);\ r_2 := \text{Loadl}(a) & \rightarrow & r_1 := \text{Loadl}(a);\ r_2 := r_1
\end{array}$$

CRF, in spite of its flexibility, still maintains a kind of store atomicity that prevents the behavior discussed in Examples 7 and 8. Intuitively, this is because the writeback operation is atomic with respect to all the processors – if a sache can retrieve the data from the memory then so can all other saches. This property will not allow us to represent certain aspects of some memory models. For example, according to the PowerPC manual [14], store operations are non-atomic and thus can exhibit the behaviors discussed in Examples 7 and 8. However, as far as we know, no implementation actually does so.

[3]We note in passing that this table is similar in style to the Table 4-20 of the IA-64 manual [12].

We can generalize the CRF model to break the atomicity of the (background) writeback operations. In such a generalized model, each site has its own memory, from where its sache can retrieve data for uncached addresses. When a sache writes a dirty copy back, it needs to write the data back to the memory of *each* site in the system, and such writeback to all the memories can occur non-atomically. With this non-atomic writeback operation, the Commit instruction guarantees that the dirty copy must have been written back to all the memories before the Commit can complete. But we do not consider such generalizations here. It is not clear if there is any practical advantage to be derived from the increased complexity except to be able to model strange corner cases of existing memory models.

3.4 Partial ordering of instructions in a CRF program

Given a sequence of CRF instructions, we can construct the exact partial order on Loadl, Storel, Commit and Reconcile instructions implied by the sequence using the following two-pass algorithm.

- **Pass One:** Traverse the sequence from the beginning to the end, introducing partial order edges for each instruction as follows:

  Case Loadl(a)**:** build edges from the previous Storel(a) and Reconcile(a), if any, to this instruction;

  Case Storel(a)**:** build edges from the previous Loadl(a) and Storel(a), if any, to this instruction;

  Case Commit(a)**:** build an edge from the previous Storel(a), if any, to this instruction;

  Case Reconcile(a)**:** build edges from the previous Fence$_{\mathrm{rr}}$(-,a), Fence$_{\mathrm{wr}}$(-,a), if any, to this instruction;

  Case Fence$_{\mathrm{rr}}$(a,-) **or** Fence$_{\mathrm{rw}}$(a,-)**:** build an edge from the previous Loadl(a), if any, to this instruction;

  Case Fence$_{\mathrm{wr}}$(a,-) **or** Fence$_{\mathrm{ww}}$(a,-)**:** build an edge from the previous Commit(a), if any, to this instruction.

- **Pass Two:** Since fences have no purpose other than to constrain the partial order between other CRF instructions, we can remove the fences by redirecting all edges coming into a fence to the instructions connected by the outgoing edges from the fence.

The resulting graph is unique after all the edges that can be derived by transitivity are removed. Figure 6 shows an example of applying this procedure. This partial order will make it clear in the next section how other memory models can be treated as special cases of CRF.

4 Universality of CRF

4.1 Upward compatibility of CRF

Suppose a company wants to adapt CRF as the memory model of their next generation microprocessors. For the sake of upward binary compatibility, the company can introduce all the CRF instructions as new instructions without replacing any of the old ones. What we show in this section is that each Load and Store (and Membar or Membar-like instructions, if any) in a program can be translated into a sequence of CRF instructions, such that exactly those behaviors that were

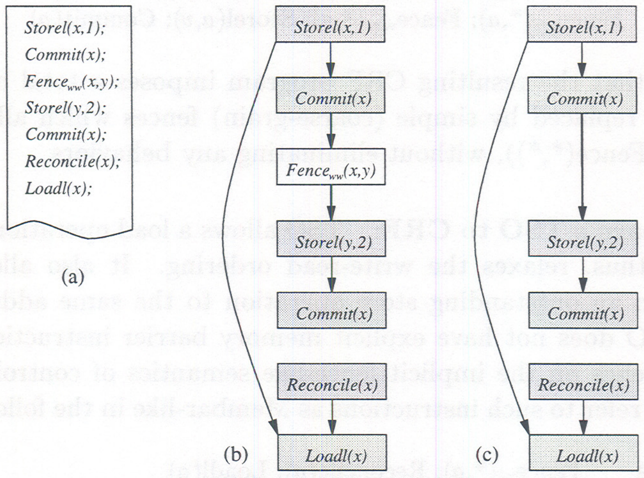

Figure 6: Partial Order of CRF Programs

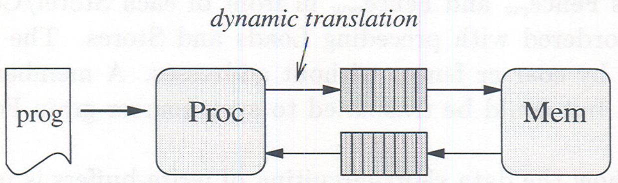

Figure 7: Dynamic Translation into CRF Programs

permitted by the source memory model are permitted by the translated program. The translations are simple enough so they can be done dynamically immediately after the decode stage.

Note that a highly constrained program, where no instruction can be reordered and where each stored value is immediately committed and each load is always preceded by a reconcile, is guaranteed to be correct. The challenge is to introduce as few reordering constrains as possible, so that the underlying architectural capabilities can be exploited to the maximum and none of the legal behaviors are ruled out.

Sequential consistency requires all the instructions to be executed in order and the execution of each instruction to appear atomic with respect to all the processors. It is tempting to think that weaker memory models such as Sparc's TSO, PSO and RMO can be expressed simply as relaxing the instruction ordering of SC [4]. This is not so because all these models break the atomicity of stores slightly by allowing the processor executing the load to see its effect before other processors. CRF captures these nuances by splitting the Store into Storel followed by a Commit and Load into a Loadl followed by a Reconcile, and controlling their reordering via fences.

In the following we give translations from SC and various other weaker memory models including IA-64 into CRF programs. The main idea is to explicitly insert the fences, which are implied by a model.

Translation from SC to CRF: This translation is simple because SC requires strict sequential execution.

$\text{Load}_{\text{sc}}(a)$	$\equiv$	$\text{Fence}_{\text{rr}}(*,a)$; $\text{Fence}_{\text{wr}}(*,a)$; Reconcile($a$); Loadl($a$)
$\text{Store}_{\text{sc}}(a,v)$	$\equiv$	$\text{Fence}_{\text{rw}}(*,a)$; $\text{Fence}_{\text{ww}}(*,a)$; Storel($a$,$v$); Commit($a$)

It is easy to show that the resulting CRF program imposes a total order on instructions. In fact, all fences can be replaced by simple (coarse-grain) fences which allow no reordering of any reads and writes (i.e., Fence(*,*)), without eliminating any behaviors.

Translation from Sparc's TSO to CRF: TSO allows a load operation to overtake outstanding store operations and thus, relaxes the write-read ordering. It also allows a load operation to retrieve the value from an outstanding store operation to the same address, thus breaking store atomicity. Sparc's TSO does not have explicit memory barrier instructions to enforce write-read ordering but instead relies on the implicit fence-like semantics of control and other read-modify-write instructions. We refer to such instructions as Membar-like in the following translation scheme.

$\text{Load}_{\text{tso}}(a)$	$\equiv$	$\text{Fence}_{\text{rr}}(*,a)$; Reconcile($a$); Loadl($a$)
$\text{Store}_{\text{tso}}(a,v)$	$\equiv$	$\text{Fence}_{\text{rw}}(*,a)$; $\text{Fence}_{\text{ww}}(*,a)$; Storel($a$,$v$); Commit($a$)
$\text{Membar-like}_{\text{tso}}$	$\equiv$	$\text{Fence}_{\text{wr}}(*,*)$

The translation scheme puts a Fence_{rr} in front of each Reconcile/Loadl pair to ensure load-load ordering, and puts Fence_{rw} and Fence_{ww} in front of each Storel/Commit pair to ensure that the Store cannot be reordered with preceding Loads and Stores. The behavior is not affected if the fences are replaced by coarser fences without addresses. A membar-like instruction is simply translated to a Fence_{wr} but could be translated to even coarser-grain Fence(*,*) without affecting the behavior.

Now let us look at how the data short-circuiting of write-buffers is modeled here. Consider the instruction sequence "Store(x,1);Store(y,2); Load(x)". According to the above rules this program is translated into the following CRF program:

$\text{Fence}_{\text{rw}}(*,x)$; $\text{Fence}_{\text{ww}}(*,x)$; Storel($x$,1); Commit($x$);
$\text{Fence}_{\text{rw}}(*,y)$; $\text{Fence}_{\text{ww}}(*,y)$; Storel($y$,2); Commit($y$);
$\text{Fence}_{\text{rr}}(*,x)$; Reconcile($x$); Loadl($x$);

If redundant fences are removed, this program is the same program as the one in Figure 6(a). According to the partial order in Figure 6(c), after the Storel(x,1) completes, the Reconcile(x) can complete with the dirty copy in the sache. Consequently, Loadl(x) can read the data from the local sache before the Commit(x) is performed, i.e., the dirty data is written back to the memory. This captures exactly the idea of data short-circuiting from the write-buffers.

Translation from Sparc's PSO to CRF: PSO, like TSO, allows a load operation to overtake store operations, and in addition, a store operation to overtake other store operations. PSO allows the write-buffers to be non-FIFO, which lets the processor merge multiple store operations in the write-buffer into one burst bus transaction. The following translation introduces appropriate fences to capture implicit load-load and load-store orderings of PSO.

$\text{Load}_{\text{pso}}(a)$	$\equiv$	$\text{Fence}_{\text{rr}}(*,a)$; Reconcile($a$); Loadl($a$)
$\text{Store}_{\text{pso}}(a,v)$	$\equiv$	$\text{Fence}_{\text{rw}}(*,a)$; Storel($a$,$v$); Commit($a$)
$\text{Membar}_{\text{pso}}$	$\equiv$	$\text{Fence}_{\text{wr}}(*,*)$; $\text{Fence}_{\text{ww}}(*,*)$

Notice short-circuiting is still permitted as in TSO.

Translation from RMO to CRF: RMO, like CRF, allows memory accesses to be reordered arbitrarily, provided that data dependencies and memory barriers are respected.

$\text{Load}_{\text{rmo}}(a)$	$\equiv$	$\text{Reconcile}(a);\ \text{Loadl}(a)$
$\text{Store}_{\text{rmo}}(a,v)$	$\equiv$	$\text{Storel}(a,v);\ \text{Commit}(a)$
$\text{Membar\#LoadLoad}_{\text{rmo}}$	$\equiv$	$\text{Fence}_{\text{rr}}(*,*)$
$\text{Membar\#LoadStore}_{\text{rmo}}$	$\equiv$	$\text{Fence}_{\text{rw}}(*,*)$
$\text{Membar\#StoreLoad}_{\text{rmo}}$	$\equiv$	$\text{Fence}_{\text{wr}}(*,*)$
$\text{Membar\#StoreStore}_{\text{rmo}}$	$\equiv$	$\text{Fence}_{\text{ww}}(*,*)$

There is a slight difference between the RMO model defined in the CRF framework, and the RMO model defined in Sparc V9's manual. In CRF, memory fences can be ordered arbitrarily with respect to each other. In contrast, according to the Sparc-V9 manual, memory barriers must be discharged in order. It is not clear if requiring in-order execution of memory barriers makes any semantic difference.

The translations from Alpha, PowerPC, and MIPS [19] memory models are similar.

Translation from IBM 370 to CRF: Like TSO, IBM 370 allows the use of write-buffers so that a load operation can be performed in case of outstanding store operations. However, it prohibits data short-circuiting from the write-buffers and requires that each load operation retrieve the data directly from the memory. In other words, the value of a store operation cannot be observed by any processor before the data becomes observable to all the processors.

The translation from IBM 370 to CRF is same as the translation from TSO to CRF, except for one extra $\text{Fence}_{\text{wr}}(a,a)$ to ensure the ordering between a Store and a Load to the same address. *Notice this effect cannot be obtained without a fine-grain fence!*

$\text{Load}_{370}(a)$	$\equiv$	$\text{Fence}_{\text{wr}}(a,a);\ \text{Fence}_{\text{rr}}(*,a);\ \text{Reconcile}(a);\ \text{Loadl}(a)$
$\text{Store}_{370}(a,v)$	$\equiv$	$\text{Fence}_{\text{rw}}(*,a);\ \text{Fence}_{\text{ww}}(*,a);\ \text{Storel}(a,v);\ \text{Commit}(a)$
Membar-like_{370}	$\equiv$	$\text{Fence}_{\text{wr}}(*,*)$

It can be seen from Figure 6, that the effect of this extra fence would be to introduce an extra edge from Commit(x) to Reconcile(x) thus preventing the execution of Loadl(x) until Storel(x) has been committed.

Translation from IA-64 to CRF: The IA-64 architecture supports a weak memory model where a processor observes all data dependencies, anti-dependencies and output dependencies. The following is a direct quote from Section 4.4.7 of the IA-64 manual [12]:

> *Memory data access ordering must satisfy read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) data dependencies to the same memory location. In addition, memory writes and flushes must observe control dependencies. Except for these restrictions, reads, writes, and flushes may occur in an order different from the specified program order. Note that no ordering exists between instruction accesses and data accesses or between any two instruction accesses. The mechanisms described below are defined to enforce a particular memory access order. In the following discussion, the terms "previous" and "subsequent" are used to refer to the program specified order. The term "visible" is used to refer to all architecturally visible effects of performing a memory access (at a minimum this involves reading or writing memory).*
>
> *Memory accesses follow one of four memory ordering semantics; unordered, release, acquire or fence. Unordered data accesses may become visible in any order. Release*

data accesses guarantee that all previous data accesses are made visible prior to being made visible themselves. Acquire data accesses guarantee that they are made visible prior to all subsequent data accesses. Fence operations combine the release and acquire semantics into a bi-directional fence, i.e., they guarantee that all previous data accesses are made visible prior to any subsequent data accesses being made visible.

Notice that IA-64 description never refers to caches but does imply that loads and stores are split-phase operations (otherwise what does it mean to wait until previous data access have become visible). The notion of store completion remains vague and informal. The issue of store atomicity (as discussed in Section 2) is also not addressed. For example, does a data value ~~have~~ has to become globally visible in a lock-step manner? Assuming stores are atomic, then the translation into CRF instructions below precisely capture their semantics.

Suppose I represents an IA-64 memory access instruction. Then mnemonics I, I.acq, I.rel represent the unordered, acquire and release form of the instruction I. Here I can be a `load`, `check load`, or `store` instruction. The semantics of the important variants of load and store instructions can be defined as follows:

Load(a)	$\equiv$	Reconcile(a); Loadl(a)
Load.acq(a)	$\equiv$	Reconcile(a); Loadl(a); $\text{Fence}_{rr}(a,*)$; $\text{Fence}_{rw}(a,*)$
Load.rel(a)	$\equiv$	$\text{Fence}_{rr}(*,a)$; $\text{Fence}_{wr}(*,a)$; Reconcile(a); Loadl(a)
Store(a,v)	$\equiv$	Storel(a,v); Commit(a)
Store.acq(a,v)	$\equiv$	Storel(a,v); Commit(a); $\text{Fence}_{wr}(a,*)$; $\text{Fence}_{ww}(a,*)$
Store.rel(a,v)	$\equiv$	$\text{Fence}_{rw}(*,a)$; $\text{Fence}_{ww}(*,a)$; Storel(a,v); Commit(a)

The semantics of check load instructions can be defined in a similar way as the load instructions. In IA-64, explicit memory ordering can also be enforced by semaphore instructions (cmpxchg, xchg and fetchadd), or memory fence (mf).

4.2 Downward Compatibility of CRF

Most existing multiprocessor systems can be interpreted as specific implementations of CRF, though more efficient implementations are possible. In this section, we show how CRF programs can be translated into programs that can run efficiently on existing microprocessors that support SC or other weaker memory model. Since the load and store instructions of all the target machines implicitly specify Reconciles and Commits, respectively, the following rules are shared by all translations.

Loadl(a)	$\equiv$	Load(a)
Storel(a,v)	$\equiv$	Store(a,v)
Commit(a)	$\equiv$	Nop
Reconcile(a)	$\equiv$	Nop

In the following we show how CRF fences should be translated into the target machine instructions.

Translating CRF Programs into SC Programs: Since memory accesses are executed strictly in-order in an SC machine, all CRF fences simply become Nop's.

Translating CRF Programs into TSO Programs: The memory fences Fence_{rr}, Fence_{rw} and Fence_{ww} are translated into Nop's, because the load-load, load-store and store-store orderings are implicitly guaranteed in TSO. A Fence_{wr} is translated into a Membar-like instruction.

Translating CRF Programs into IBM 370 Programs: Since the IBM 370 model is slightly stricter than the TSO model, the translation can avoid some memory barriers by taking advantage from the fact that IBM 370 prohibits data short-circuiting from write-buffers. This suggests that a Fence$_{wr}$ with identical pre- and post-address can be translated into a Nop.

Translating CRF Programs into PSO Programs: The Fence$_{rr}$ and Fence$_{rw}$ are translated into Nop's since the load-load and load-store orderings are implicitly preserved in PSO. Both Fence$_{wr}$ and Fence$_{ww}$ are translated into Membar.

Translating CRF Programs into RMO Programs: RMO assumes no ordering between memory accesses except for data dependencies. The translation scheme from CRF to RMO translates each memory fence into the corresponding coarser-grain memory barrier instruction as follows:

Fence$_{rr}(a_1,a_2)$	$\equiv$	Membar#LoadLoad
Fence$_{rw}(a_1,a_2)$	$\equiv$	Membar#LoadStore
Fence$_{wr}(a_1,a_2)$	$\equiv$	Membar#StoreLoad
Fence$_{ww}(a_1,a_2)$	$\equiv$	Membar#StoreStore

Translating CRF Programs into IA-64 Programs: The translation from CRF programs into IA-64 programs is also straightforward. As with previous translation schemes, Commit and Reconcile become Nops because such semantics will be covered by IA-64's Load and Store instructions. Each Loadl or Storel is translated into a Load or Store, respectively. The only subtle issue is for fence instructions. While we can always translate a CRF fence into a memory fence (mf) instruction, we can achieve the same semantics by using the ordered load and store instructions. For example, a Fence$_{rw}(a_1,a_2)$ instruction becomes a Nop and the preceding Loadl(a_1) instruction (if such instruction does not exist, then the fence can be treated as a nop) is translated as a Load.acq instruction (instead of an unordered Load instruction). Alternatively, the following Storel(a_2) can be translated as a Store.rel instruction (instead of an unordered Store instruction).

It is obvious that, with proper look-ahead (or look-behind) capability, these methods can be made into an on-line translation scheme from CRF into IA-64. When the fence is beyond the look-ahead scope, an explicit memory fence instruction can always be used. This demonstrates the downward compatibility of CRF with respect to IA-64.

5 Conclusion

The CRF model exposes both instruction reordering and caches to the instruction set architecture. Commercial microprocessor designers have accepted the fact that instruction reordering should be exposed, as seen by the growing acceptance of various types of fence instructions. But there is a great effort to keep all forms of data replication transparent while acknowledging the split-phase nature of load and store operations. A transparent cache implies store atomicity but store atomicity does not imply transparent caches. Indeed the way the CRF model exposes caches preserves store atomicity. That is, as soon as one consumer processor can see the value stored by a producer, so will all other processors.

Carefully exposing caches, as done in CRF, has three consequences. The meaning of all instructions is local. That is, an instruction does not have to wait for the actions of another processor before it can complete. At most, there is some interaction with a global memory. A second consequence is that caches are transparent for sequential codes executed on uniprocessors. The same ISA and same memory model are suitable for uniprocessor, SMP, and DSM machines. Finally, unlike the behavior of programs with data races on machines with weak memory models, the meaning of all CRF programs is all well-defined.

The universality of CRF, as shown in this paper, has several consequences as well. CRF provides a way to evolve microarchitectures without having to make continuous adjustments to the programming model. Compilers can adopt CRF as a universal interface and then trivially translate the CRF program into an ISA with a specific memory model. Furthermore, if the compiler knows some properties of a program, such as the data-race-free property, it can compile better CRF programs. Along the same lines, cache coherent protocols for CRF, such as Cachet [15], are automatically applicable to any memory model whose programs can be expressed in CRF.

Finally, it is important to understand that this paper is less a criticism of any particular instruction set architecture than it is of the imprecision in the descriptions of memory related instructions. Adoption of CRF may lead not only to precise definitions of existing instruction sets but may also lead us into a world where one can add architectural optimizations safely.

Research Credit: This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by grants from International Business Machines Corporation (IBM) and the National Science Foundation award CCR-9970735.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] Sarita V. Adve and Mark D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th International Symposium On Computer Architecture*, pages 2–14, June 1990.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] William W. Collier. *Reasoning About Parallel Architecturers* . Prentice Hall, Englewood Cliffs, NJ, 1992.

[5] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13rd International Symposium On Computer Architecture*, pages 434–442, June 1986.

[6] Kourosh Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Phd. thesis, Stanford University, 1995.

[7] 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 International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta*, May 1999.

[9] Mark D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, August 1998.

[10] L Iftode, J P Singh, and Kai Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[11] Intel, editor. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual.* Intel Corporation, 1996.

[12] Intel, editor. *IA-64 Application Developer's Architecture Guide.* Intel Corporation, May 1999.

[13] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium On Computer Architecture*, pages 13–21, May 1992.

[14] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* Morgan Kaufmann, 1994.

[15] Xiaowei Shen, Arvind, and Larry Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[16] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta*, May 1999.

[17] Richard L. Sites and Richard T. Witek, editors. *Alpha AXP Architecture Reference Manual (Second Edition).* Butterworth-Heinemann, 1995.

[18] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual (Version 9.* Prentice-Hall, 1994.

[19] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, April 1996.