LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers

Computation Structures Group Memo 413
October 1998 (Revised: February 1999)

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

To appear in Proceedings of the 26th International Symposium on Computer Architecture, Atlanta, Georgia, May 1999.

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 the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers

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

Abstract

We present a new mechanism-oriented memory model called Commit-Reconcile & Fences (CRF) and define it using algebraic rules. Many existing memory models can be described as restricted versions of CRF. The model has been designed so that it is both easy for architects to implement, and stable enough to serve as a target machine interface for compilers of high-level languages. The CRF model exposes a semantic notion of caches (saches), and decomposes load and store instructions into finer-grain operations. We sketch how to integrate CRF into modern microprocessors and outline an adaptive coherence protocol to implement CRF in distributed shared-memory systems. CRF offers an upward compatible way to design next generation computer systems.

1. Loads and Stores: The CISC of Nineties

Caching and instruction reordering are ubiquitous features of modern computer systems and are necessary to achieve higher performance. For uniprocessor configurations, these features are mostly transparent and exposed only for some low-level memory-mapped input/output operations. For multiprocessor configurations however, these features are anything but transparent. Indeed, a whole area of research has evolved around what view of memory should be presented to the programmer, the compiler writer, and the computer architect.

Every programming language has a memory model, regardless of whether it is described explicitly or not (*e.g.*, programmer-centric models [16, 5, 19]). It is the task of the compiler to ensure that the semantics of a high-level program is preserved when its compiled version is executed on an architecture with a certain low-level memory model (*e.g.*, architecture-centric models [25, 18, 26, 14]). The essence of any memory model is the correspondence between each load instruction and the store instruction that supplies the value retrieved by the load. Unfortunately, at the architecture level, memory access operations often have some sophisticated implementation characteristics that make it difficult to specify the resulting memory model precisely.

Our approach is to decompose the load and store instructions into finer-grain orthogonal operations and use them to define the Commit-Reconcile & Fences (CRF) memory model. The CRF model has a semantic notion of caches, referred to as *saches*, which makes the operational behavior of data replication to be part of the model. Both loads and stores are performed directly on local saches and new instructions are provided to move data between saches and memory whenever necessary. The Commit instruction ensures that a modified value in the sache is written back to the memory, while the Reconcile instruction ensures a stale value is purged from the sache. CRF also provides fine-grain fences to control the reordering of memory related instructions. Thus, a normal load or store instruction can be thought of as follows:

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

where Loadl and Storel represent loading data from and storing data to the local sache, respectively.

CRF is a *mechanism-oriented* memory model and intended for architects and compiler writers rather than for high-level parallel programming (see Figure 1). It is defined by giving precise (algebraic) semantics to the memory related instructions so that every CRF program has a well-defined operational behavior. The CRF mechanisms give architects great flexibility for efficient implementations, and at the same time these mechanisms give compiler writers all the control they need. A compiler can move around or even eliminate some Commit, Reconcile and Fence operations in a program.

There are many benefits to our approach. CRF permits aggressive cache coherence protocols in distributed shared-memory (DSM) systems because no operation explicitly

Programmer-centric Models				
Sequential Consistency	Release Consistency	Properly-labeled Programs	Cilk's Model	Java's Model

Commit-Reconcile & Fences (CRF)

Implementations				
Sequential Consistency	Sparc's Models	Alpha's Model	PowerPC's Model	CRF Processors

Figure 1. CRF: A Memory Model for Architects and Compiler Writers

or implicitly involves multiple saches. Furthermore, any cache-coherence protocol for CRF is automatically a correct protocol for all other memory models whose programs can be transformed into CRF programs.

In addition, there is no need to distinguish between ordinary variables and synchronization variables used as locks. In fact, the choice and implementation of synchronization mechanisms is an orthogonal issue. CRF mechanisms can be incorporated in stages in future systems without loss of compatibility with existing systems.

The salient features of CRF may be described as follows:

- It is easy to translate a variety of high-level programming models into CRF. Translation of programs based on models such as release consistency into CRF is straightforward.
- Most existing multiprocessor systems can be interpreted as specific implementations of CRF, though more efficient implementations are possible.
- In CRF, stores to each memory location are totally ordered so that they are always observed in the same order by all the processors. This is a deliberate design choice to avoid semantic complications without compromising implementation flexibility.
- The set of algebraic rules, that define all legal behaviors according to the CRF model, can be used by architects and compiler writers to design and verify the correctness of their optimizations.

Paper organization: We briefly discuss existing memory models in Section 2 before introducing our formalism in Section 3. After presenting the CRF model in Section 4, we discuss the relationship of CRF with some existing memory models in Section 5. In Section 6, we sketch an adaptive cache coherence protocol that implements CRF. Section 7 discusses the potential impact of CRF on microarchitectures, and some conclusions follow in Section 8.

2. Weaker Memory Models

Sequential consistency (SC) has been the dominant memory model in parallel computing for decades due to its simplicity [16] and is thus the standard against which other models must be compared. SC requires that memory accesses of a program be performed in-order on each processor and be atomic with respect to each other and is thus clearly at odds with both instruction reordering and caching. Ingenious solutions have been devised to keep both of these features transparent so that at the high-level, a programmer assuming SC cannot detect if and when the memory accesses are out-of-order or non-atomic. For small scale parallel machines, it is well understood how to preserve the access atomicity of SC with cache coherence protocols, and recent advances in the use of speculative execution permit reordering of loads without destroying the sequentiality of SC.

The desire to achieve higher performance has led to various relaxed or weaker memory models [21, 12, 15, 8, 6]. Broadly speaking, weaker memory models weaken either the sequentiality constraint or the atomicity constraint of SC. The *weak-ordering* property allows certain memory accesses to be performed in a different order than the program order unless explicit ordering constraints are specified. The *weak-atomicity* property exposes data replication by allowing store accesses to be performed in some non-atomic fashion. An aggressive memory model can accommodate some combination of both weak-ordering and weak-atomicity properties.

2.1. Properly Synchronized Programs

It is common practice in parallel programming to use locks to ensure that only one processor at a time can access a shared variable (though in many programs it is perfectly safe to read a shared variable without acquiring the lock). A data race occurs when there are multiple concurrent accesses to a shared variable, at least one of which is a write. Informally, a *properly synchronized* program has no data races; races are limited to acquiring locks.

Although it is generally undecidable if a program is properly synchronized, it is relatively easy for the programmer to characterize each memory operation as ordinary or synchronization access. Synchronization accesses can be further classified as acquire and release operations, loop and non-loop operations, and so on. Based on such classifications, the notions of data-race-free programs [2, 3] and properly-labeled programs [10, 9] have been defined. In each definition, conflicting ordinary accesses are separated (ordered) by synchronization accesses. For properly synchronized programs, SC behavior can be achieved on an architecture with an appropriate weaker memory model [1].

2.2. Models with Weak-Ordering

Modern microprocessors support memory models with weak-ordering to hide latency for performance improvement. Memory fences are provided at the programming level to ensure proper ordering constraints between specific memory accesses whenever necessary. Examples of memory fences include the Membar instruction in Sparc [26] and the Sync instruction in PowerPC [18]. Synchronization instructions such as Test-&-Set and Swap often act as fences. IBM 370's conditional instructions have a fence-like effect, as does PowerPC's EIEIO instruction.

Different weak-ordering models allow memory accesses to be reordered under different conditions. In Sparc, Total Store Order (TSO) allows a load instruction to be performed before outstanding store instructions complete, which virtually models FIFO write-buffers. The Partial Store Order (PSO) model further allows stores to be reordered, so that stores to the same cache line can be merged in write-buffers. Sparc-v9 defined Relaxed Memory Order (RMO) in which loads and stores can be performed in arbitrary order, provided that the so-called self-consistency is preserved. In each of these cases, the programmer must know the memory model of the underlying architecture so that when necessary, he can insert appropriate instructions to ensure that SC assumptions are not violated.

2.3. Models with Weak-Atomicity

Although weak-ordering models allow memory accesses to be performed out-of-order, they still require each memory access to be atomic with respect to other memory accesses. The semantic effect of each store operation must be observable by other processors in a lock-step. In the presence of caches, some mechanism is needed to prevent other processors from observing stale values in their caches. This is usually accomplished by invalidating all outstanding copies of the address in other caches. Invalidation increases store latencies and can cause dramatic performance degradation. In the past, such delays have been tolerable in small SMP's using snoopy bus protocols, but this may not be so in future.

Release consistency (RC) allows non-atomic memory accesses since the execution of memory accesses between acquire and release operations does not have to be visible immediately to other processors [12, 17]. The essence of RC is that memory accesses before a release must be globally performed before the synchronization lock can be released. Lazy release consistency (LRC) goes a step further; it allows a synchronization lock to be released to another processor even before previous memory accesses have been globally performed, provided the semantic effect of those memory accesses has become observable to the processor about to acquire the lock [15]. Again it can be shown that properly synchronized programs execute correctly under both RC and LRC, giving more flexibility in implementations.

Weak memory models have often eluded precise definitions, a fact that causes complications when there is a multi-level, non-uniform memory hierarchy with a mixture of shared buses and networks.

3. Specifying a Memory Model

3.1. Program Order and Memory Models

Memory models are often defined based on the concept of program order, which is an execution trace of memory accesses. The program order has been defined in various ways; the following definition from "The Sparc Architecture Manual (Version 9)" is typical:

> *A program order execution trace is an execution trace that begins with a specified initial instruction and executes one instruction at a time in such a fashion that all the semantic effects of each instruction take effect before the next instruction is begun. The execution trace this process generates is defined to be the program order. ... Program order specifies a unique total order for all memory transactions initiated by one processor.*

We find such definitions defective on two counts. First, it is not possible to define program order without first specifying the memory model, because the memory model affects the program order. The following example illustrates the problem:

	Processor 1	Processor 2
	r = Load(a_1);	Store(a_1,1);
	Jz(r,L2);	...
L1:	Store(a_2,100);	Store(a_1,0);
	Jz(r, L3);	
L2:	Store(a_2,200);	
	Jz(r,L1);	
L3:	...	

Assume initially both memory locations a_1 and a_2 contain value 0. What is the program order for processor 1 in regards to the two stores at L1 and L2? If register r gets value 0 then L2 is executed before L1; otherwise L1 is executed before L2. Since this depends on the value retrieved by the load instruction, it cannot be determined without a memory model. *Program order is not a well-defined concept without the inclusion of a memory model.*

The second problem with the program order definition is that, by insisting on executing one instruction at a time, many legal and interesting memory behaviors cannot be observed. Consider the following program:

Processor 1		Processor 2
Store(a_1,1);	L:	r_2 = Load(a_2);
r_1 = Load(a_1);		Jz(r_2,L);
Store(a_2,r_1);		r = Load(a_1);

Suppose initially locations a_1 and a_2 contain value 0. Processor 1 eventually stores value 1 to location a_2, while processor 2 loops until the value of location a_2 becomes non-zero (*i.e.*, 1). Let us assume that loads cannot be reordered (or equivalently, the memory model does not allow loads to be reordered). Is it possible for register r to get value 0? Whether r gets value 0 or 1 can affect the program order on processor 2; just imagine the code of processor 1 from the previous example replaces the last instruction of processor 2 here. Some memory models allow the two stores on processor 1 to be reordered in spite of apparent data dependencies. For example, register r_1 can get value 1 before the store to location a_1 is "globally performed" due to the use of write-buffers. This cannot happen if instructions are executed one at a time as dictated by the definition of the program order.

Our way of defining memory models side-steps the pitfalls associated with definitions based on program order.

3.2. Memory Model as an Input-Output Relation

The observable behavior of a program on a computer is determined both by its processor microarchitecture (*e.g.*, out-of-order and speculative execution) and the memory architecture (*e.g.*, memory, caches, cache-coherence protocols). At an abstract level, it is useful to think of a computer as having two types of subsystems – processor and memory. Each processor generates a stream of memory requests, for which the memory produces a stream of replies. The memory system behaves as an oracle that defines a value for each load request generated during a program execution. A *memory model* is the specification of such an oracle. For parallel systems, memory models are invariably nondeterministic, that is, for the same set of input streams a range of behaviors are acceptable as output.

We will present memory models as a mathematical relation between processor request streams and memory reply streams. A request stream consists of loads and stores, and in weaker models, some special instructions like fences. We assume processors attach a unique transaction tag to each request and the memory system generates a corresponding reply using the same transaction tag. The replies are not necessarily generated in the same order in which the requests are processed. In case of a load request, the reply contains the value returned from the memory. For other types of requests the reply simply contains an acknowledgment (Ack). The order of requests in an input stream is significant and represents the order imposed on the requests by the program execution. There is no order on replies in the output stream. Generally, many different sets of output streams represent legal replies for the same set of input streams.

Our memory model definitions can be used to design and verify cache coherence protocols and processor optimizations. However, any issue regarding program behavior requires both a processor model and a memory model and is discussed in this paper only tangentially. (We have discussed program behavior issues related to microarchitectures with register renaming and speculative executions elsewhere [22]).

In the rest of this section, we introduce our TRS formalism and use it to model SC.

3.3. Term Rewriting Systems

We will use Term Rewriting Systems (TRS's) to define memory models. A TRS consists of a set of terms and a set of rewriting rules. The terms represent system states and the rules specify state transitions. The general structure of rewriting rules is as follows:

$$\begin{array}{ll} \quad s_1 & \text{if } p(s_1) \\ \to s_2 & \end{array}$$

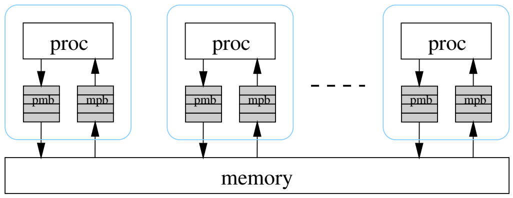
where s_1 and s_2 are terms and $p(s_1)$ is an optional predicate about s_1.

A rule can be used to rewrite a term if its left-hand-side pattern matches the term or one of its subterms, and the corresponding predicate, if any, is true. If several rules are applicable, then any one of them may be applied. If no rule is applicable, then the term cannot be rewritten any further. A rewriting strategy can be used to specify which rule among the applicable rules should be applied at each rewriting step.

Notation: We use '[]' as the meta notation in grammars to separate disjuncts. It is important to distinguish between variables and constants while pattern matching. A variable matches any expression while a constant matches only itself. We will follow the convention where variables and constants are represented by identifiers that begin with a lower-case and upper-case letter, respectively. Connectives such as ';' are also constants. We use 'ϵ' to represent the empty term (*e.g.*, an empty cache), and '-' the wild-card term that can match any term. We use '|' as a connective to indicate that ordering does not matter (*i.e.*, '|' is associative and commutative).

3.4. Example: Sequential Consistency

As an example, we use a TRS to define SC. The system is modeled as a memory and a set of sites (see Figure 2). Each site contains a processor, a processor-to-memory buffer (pmb) and a memory-to-processor buffer (mpb). A memory request is a Load or Store instruction. SC can be defined by the following rules:



SYS	≡	Sys(MEM, SITES)
SITES	≡	SITE [] SITE | SITES
SITE	≡	Site(PMB, MPB, PROC)
PMB	≡	ϵ [] ⟨t,REQ⟩;PMB
MPB	≡	ϵ [] ⟨t,REP⟩|MPB
REQ	≡	Load(a) [] Store(a,v)
REP	≡	v [] Ack

Figure 2. Semantic Configuration of SC

SC-Load Rule

Sys(m, Site(⟨t,Load(a)⟩;pmb, mpb, p) | sites)
→ Sys(m, Site(pmb, mpb|⟨t,m[a]⟩, p) | sites)

SC-Store Rule

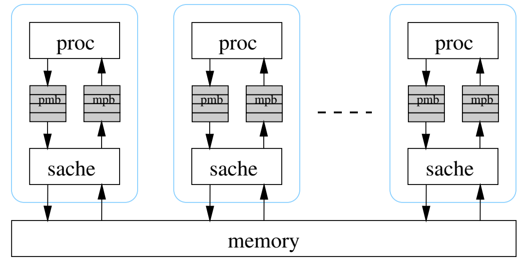
Sys(m, Site(⟨t,Store(a,v)⟩;pmb, mpb, p) | sites)
→ Sys(m[a:=v], Site(pmb, mpb|⟨t,Ack⟩, p) | sites)

where m[a] refers to the value of memory location with address a, and m[a:=v] represents memory m with location a updated with value v. Since only the instruction at the front of pmb can be executed, memory requests are processed in-order on each processor. Memory accesses are semantically atomic with respect to one another because there is no data replication. Since the connective '|' implies no ordering, any site can be brought to the leftmost position in the site group. Thus, if two processors intend to access the same address, either can proceed.

The above two rules completely define all possible outcomes for a given set of request streams for the SC model. We will now show that, even without a processor model, this definition can be used to decide the correctness of some optimizations for microarchitectures.

3.5. Some Optimization Rules

Suppose each processor keeps outstanding instructions in some buffers in-order. On a load, the processor checks the buffer, and if the preceding instruction is a store to the same address, the value of the store instruction is returned immediately. Similarly, on a store, if the preceding instruction is a store to the same address, then the preceding store instruction can be discarded without writing back to the memory. The following two rules express these optimizations:



SYS	≡	Sys(MEM, SITES)
SITES	≡	SITE [] SITE | SITES
SITE	≡	Site(SACHE, PMB, MPB, PROC)
SACHE	≡	ϵ [] Cell(a,v,CS) | SACHE
CS	≡	Clean [] Dirty
PMB	≡	ϵ [] ⟨t,REQ⟩;PMB
MPB	≡	ϵ [] ⟨t,REP⟩|MPB
REQ	≡	LoadI(a) [] StoreI(a,v)
	[]	Commit(a) [] Reconcile(a)
	[]	Fence$_{rr}$(a_1,a_2) [] Fence$_{rw}$(a_1,a_2)
	[]	Fence$_{wr}$(a_1,a_2) [] Fence$_{ww}$(a_1,a_2)
REP	≡	v [] Ack

Figure 3. Semantic Configuration of CRF

SC-Load-Bypass Rule

Site(pmb$_1$;⟨t_1,Store(a,v)⟩;⟨t_2,Load(a)⟩;pmb$_2$, mpb, p)
→ Site(pmb$_1$;⟨t_1,Store(a,v)⟩;pmb$_2$, mpb|⟨t_2,v⟩, p)

SC-Store-Merge Rule

Site(pmb$_1$;⟨t_1,Store(a,v_1)⟩;⟨t_2,Store(a,v_2)⟩;pmb$_2$, mpb, p)
→ Site(pmb$_1$;⟨t_2,Store(a,v_2)⟩;pmb$_2$, mpb|⟨t_1,Ack⟩, p)

These rules are correct in the sense that, given a set of input streams, they do not add any new behavior to the set of behaviors generated by the SC rules (or equivalently, the SC rules can simulate the behaviors generated by the optimization rules). Of course, given a processor model, addition of such rules may affect the set of behaviors observable for a program. However, all such program behaviors would correspond to some behavior permissible by the SC model.

4. The CRF Memory Model

CRF exposes both data replication and instruction reordering at the programming level. Each site has a semantic cache (sache), on which LoadI (load-local) and StoreI (store-local) instructions operate (see Figure 3). The model assumes memory accesses can be reordered as long as data dependence constraints are preserved, and provides memory fences to enforce ordering if needed.

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

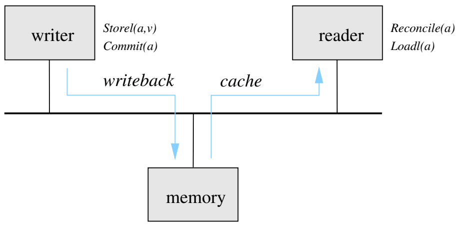


Figure 4. Memory Rendezvous of CRF

the reader: the writer performs a Commit operation to guarantee that the 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 subsequent load operations must retrieve the data from the memory (see Figure 4).

Semantic caches do not necessarily correspond to caches in implementations; they are needed purely for semantic reasons. Similarly, the rendezvous between the writer and reader does not have to be the main memory; a cache coherence protocol may use any cache in the memory hierarchy as the rendezvous point. The only thing that matters is that any system that implements CRF must maintain the same observable memory behavior.

The definition of CRF includes two sets of rules. The first set of rules specifies the execution of Loadl, Storel, Commit and Reconcile instructions. It also includes rules that govern the data propagation between semantic caches and memory. The second set of rules deals with instruction reordering and memory fences. We also refer to the first set of rules as *the Commit-Reconcile (CR) model* because these rules by themselves define a memory model, which is the same as the CRF model except that instructions are executed strictly in-order.

4.1. The Commit-Reconcile Model

There are two states for sache cells, Clean and 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. Notice in CRF, different saches can have a cell with the same address but different values.

Loadl and Storel Rules: A Loadl or Storel can be performed if the address is cached in the sache:

CRF-Loadl Rule
Site(sache, ⟨t,Loadl(a)⟩;pmb, mpb, p)
if Cell(a,v,-) ∈ sache
→ Site(sache, pmb, mpb|⟨t,v⟩, p)

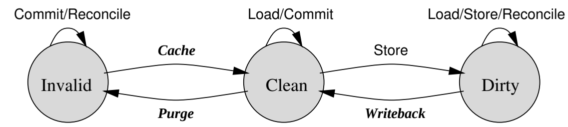


Figure 5. Sache State Transitions of CRF

CRF-Storel Rule
Site(Cell(a,-,-) | sache, ⟨t,Storel(a,v)⟩;pmb, mpb, p)
→ Site(Cell(a,v,Dirty) | sache, pmb, mpb|⟨t,Ack⟩, p)

Although the store rule above requires that the address be cached before the Storel can be performed, it makes no semantic difference to allow the Storel to be performed even if the address is not cached. This is because, if the address is not cached, the sache can first obtain a Clean copy from the memory (by applying the cache rule given below), and then perform the Storel access. This can be represented by a straightforward derived rule.

Commit and Reconcile Rules: On a Commit operation, if the address is cached and the cell's state is Dirty, the data must be first written back to the memory (by applying the writeback rule given below). On a Reconcile operation, if the address is cached and the cell's state is Clean, the cell must be first purged from the sache (by applying the purge rule given below).

CRF-Commit Rule
Site(sache, ⟨t,Commit(a)⟩;pmb, mpb, p)
if Cell(a,-,Dirty) ∉ sache
→ Site(sache, pmb, mpb|⟨t,Ack⟩, p)

CRF-Reconcile Rule
Site(sache, ⟨t,Reconcile(a)⟩;pmb, mpb, p)
if Cell(a,-,Clean) ∉ sache
→ Site(sache, pmb, mpb|⟨t,Ack⟩, p)

Note that a Reconcile operation can complete while the address is cached in the Dirty state. This allows proper modeling of load bypassing in write-buffers.

Cache, Writeback and Purge Rules: A sache can obtain a Clean copy from the memory, if the address is not cached at the time (thus no sache can 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. Figure 5 illustrates the sache state transitions (Invalid indicates the address is not cached).

CRF-Cache Rule
Sys(m, Site(sache, pmb, mpb, p) | sites)
if a ∉ sache
→ Sys(m, Site(Cell(a,m[a],Clean) | sache, pmb, mpb, p)
| sites)

$I_2 \Rightarrow$ $I_1 \Downarrow$	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
$\text{Fence}_{rr}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
$\text{Fence}_{rw}(a_1,a_2)$	true	$a_2 \neq a'$	true	true	true	true	true	true
$\text{Fence}_{wr}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
$\text{Fence}_{ww}(a_1,a_2)$	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 6. Instruction Reordering Table of CRF

CRF-Writeback Rule

Sys(m, Site(Cell(a,v,Dirty) | sache, pmb, mpb, p) | sites)
$\rightarrow$ Sys(m[a:=v], Site(Cell(a,v,Clean) | sache, pmb, mpb, p) | sites)

CRF-Purge Rule

Site(Cell(a,-,Clean) | sache, pmb, mpb, p)
$\rightarrow$ Site(sache, pmb, mpb, p)

These rules are also called background rules, since they can be applied even though no instruction is executed by any processor. The background rules can potentially propel optimizations that are more aggressive than conventional techniques such as non-binding prefetch.

It is worth noting that we can add extra Commit and Reconcile instructions in a program without affecting its semantics. The set of behaviors that is generated by the program with extra Commit and Reconcile instructions is a subset of the behaviors of the original program. This is because, regardless of whether a Commit or Reconcile is executed, a dirty cell can always be written back to the memory and a clean cell can always be purged from a sache.

The operational specification of CRF includes a set of *imperative rules* only; intentionally this does not address implementation issues. For example, suppose a processor executes a Commit instruction while a dirty copy is cached for the address. The processor will stall until the writeback rule is applied. In practice, proper *directive rules* must be incorporated to ensure the liveness of the system. Directive rules, however, have no semantic implications and thus, are not part of the CRF definition. More discussion about the imperative-directive design methodology can be found elsewhere [23].

4.2. The Fence Operation

CRF allows memory accesses to be reordered if they access different addresses or if they are both Loadl instructions. It provides four types of memory fences to control reordering: Fence_{rr} (read-read), Fence_{rw} (read-write), Fence_{wr} (write-read) and Fence_{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, $\text{Fence}_{rw}(a_1,a_2)$ 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 $\text{Fence}_{rw}(a_1,a_2)$ cannot be reordered.

A Fence_{wr} or Fence_{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 a Storel operation to be completed if it is not followed by a Commit. Similarly, a Fence_{rr} or Fence_{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 6 concisely defines the conditions under which two adjacent memory instructions can be reordered (assume instruction I_1 precedes instruction I_2, and a 'true' condition indicates that the reordering is allowed). 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. This rule is commutable in the sense that reordered transactions can be reordered back. Not all the reordering rules commute. For example, the rule represented by the Fence_{rr}-Loadl entry does not commute: once the reordering is performed, the transactions cannot be reordered back unless the address of the Loadl instruction and the pre-address of the Fence_{rr} instruction are different (according to the Loadl-Fence_{rr} entry).

We also need a rule to discharge a memory fence:

CRF-Fence Rule

Site(sache, $\langle$t,$\text{Fence}_{\alpha\beta}(a_1,a_2)\rangle$;pmb, mpb, p)
$\rightarrow$ Site(sache, pmb, mpb | $\langle$t,Ack$\rangle$, p)

The reordering rules are useful for the compiler writer to decide whether specific compiler transformations preserve program semantics. The specification of CRF also demonstrates a framework in which different memory models can be defined and analyzed. For example, we can ensure that all load and store operations to the same location are performed in-order on each processor by changing the LoadI-LoadI entry to "$a \neq a'$".

Again it is worth noting that adding an extra Fence in a program can eliminate a permissible behavior but never add a new behavior. In particular, a strict sequential execution of instructions always generates a legal behavior.

4.3. Coarse-grain Fences, Commits and Reconciles

We have chosen fine-grain fence, commit and reconcile operations to define the memory model but coarse-grain versions of these operations may be more practical at the instruction set level.

A coarse-grain fence imposes an ordering constraint with respect to address ranges, instead of individual locations. For example, $\mathsf{Fence}_{rw}(A_1,A_2)$ ensures that all preceding LoadI operations to address range A_1 must be performed before any following StoreI operation to address range A_2 can be performed. It can be defined in terms of $|A_1|.|A_2|$ fine-grain fences and obey all the reordering rules given earlier. Similarly, Commit(A) and Reconcile(A) can be defined in terms of fine-grain Commit and Reconcile operations, respectively. An address range may be a cache line, a page or the whole address space (represented by *).

Of particular interest are memory fences that impose ordering constraints between some memory range and an individual location. As an example, we define the following pre- and post- fences:

$$\begin{aligned} \mathsf{PreFenceW(a)} &\equiv \mathsf{Fence}_{rw}(*,a);\ \mathsf{Fence}_{ww}(*,a) \\ \mathsf{PostFenceR(a)} &\equiv \mathsf{Fence}_{rr}(a,*);\ \mathsf{Fence}_{rw}(a,*) \end{aligned}$$

Informally, PreFenceW(a) requires that all memory accesses (*i.e.*, LoadI and Commit) preceding the fence be completed before any store to location a following the fence can be performed. PostFenceR(a) requires that all loads to location a preceding the fence be completed before any memory access (*i.e.*, Reconcile and StoreI) following the fence can be performed.

5. Relationship with Other Models

There is a simple translation scheme from SC programs to CRF programs. We can augment an SC program to a CRF program by substituting each Load/Store instruction with a LoadI/StoreI instruction, placing a Reconcile before each LoadI and a Commit after each StoreI, and inserting memory fences appropriately. This translation guarantees that the augmented program in CRF has the same program behavior as the original program in SC.

A program that is crucially dependent on the SC semantics for its correctness is the Dekker's algorithm for mutual exclusion. The essence of this algorithm is that a processor first signals its intent to enter the critical section by asserting a flag (a_1 and a_2 for processors 1 and 2, respectively), and then checks whether the other processor is also trying to enter. It can enter the critical section only when the other processor has not set its flag (this part of the code is not shown below). Initially both locations a_1 and a_2 contain value 0.

Processor 1	**Processor 2**
StoreI(a_1,1);	StoreI(a_2,1);
Commit(a_1);	Commit(a_2);
$\mathsf{Fence}_{wr}(a_1,a_2)$;	$\mathsf{Fence}_{wr}(a_2,a_1)$;
Reconcile(a_2);	Reconcile(a_1);
r = LoadI(a_2);	r = LoadI(a_1);

The Lock and Unlock operations for mutual exclusion can be implemented with much less effort using synchronization instructions such as Test-&-Set, Swap or Load-Reserve/Store-Conditional. In CRF, these instructions by themselves have no ordering implication on preceding and following instructions. Memory fences can be used to enforce necessary ordering constraints, where Lock is considered to be both a LoadI and StoreI operation, and Unlock simply a StoreI operation.

We can translate programs based on release consistency to CRF by defining the Release and Acquire operations as follows:

$$\begin{aligned} \mathsf{Release(s)} &\equiv \mathsf{Commit(*);\ PreFenceW(s);\ Unlock(s)} \\ \mathsf{Acquire(s)} &\equiv \mathsf{Lock(s);\ PostFenceR(s);\ Reconcile(*)} \end{aligned}$$

This can lead to better performance (especially for DSM systems) than implementations of RC on existing microprocessors. Memory accesses after a Release can be performed before the semaphore is released, because the Release only imposes a pre-fence on preceding accesses. Memory accesses before an Acquire do not have to be completed before the semaphore is acquired, because the Acquire only imposes a post-fence on following memory accesses. In addition, modified data of store operations before a Release need to be written back to the memory at the release point, but stale data in other caches do not have to be invalidated or updated since it will be reconciled at the next acquire point.

Modern microprocessors often exhibit some relaxed memory models. They provide very coarse-grain memory fences that apply to all addresses, and have no commit/reconcile like instructions. For example, Sparc's RMO model can be represented using CRF instructions as follows:

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

Unlike CRF, according to the manual, Sparc requires that Membar instructions be applied in-order. The exact semantics of Membar instructions can be obtained by simply modifying the corresponding entries in the reordering table. However, it is not clear to us why fences must be performed in-order.

Some memory models (*e.g.*, Location Consistency [7, 8]) cannot be represented by CRF. In CRF, if the values of two stores (not necessarily from the same processor) are observed by more than one processor, then they must be observed in the same order, provided the load instructions used in the observation are executed in-order. *There is a total order on stores for each address in CRF*. For example, in the following program, if register r_1 gets value 2, then register r_2 also must get value 2.

Processor 1	**Processor 2**
Storel(a,1);	Storel(a,2);
r_1 = Loadl(a);	r_2 = Loadl(a);

6. Cache Coherence Protocols for CRF

Cachet [24], an adaptive cache coherence protocol, has been developed to implement CRF on the MIT StarT multiprocessor system [4]. Cachet is a seamless integration of a number of micro-protocols, each of which has been optimized for a different access pattern. The design is motivated by the belief that a protocol that adapts to changing access patterns should perform better than any fixed protocol. Micro-protocols are distinctive in the actions performed by the protocol engine while committing dirty cells and reconciling clean cells.

Cachet-Base: This straightforward implementation of CRF simply uses the memory as the rendezvous point. A Commit instruction for an address cached in the Dirty state requires that the modified data be written back to the memory before the instruction can complete. A Reconcile instruction for an address cached in the Clean state requires the data be purged from the cache before the instruction can complete. An attractive characteristic of Cachet-Base is its simplicity: no state needs to be maintained on the memory side.

Cachet-WriterPush: Since load operations are usually more frequent than store operations, it is desirable to allow a Reconcile to complete even when the address is cached in the Clean state. Subsequent load accesses to the address targeted by the Reconcile will then cause no cache miss. Correspondingly, when a Commit instruction is performed on an address cached in the Dirty state, the Clean copies of the address are purged from all other caches before the Commit can complete. Therefore, committing an address that is cached in the Dirty state can be a lengthy process.

Cachet-Migratory: When an address is exclusively accessed by one processor for a reasonable time period, it makes sense to give the cache the exclusive ownership so that all instructions on the address become local operations. This is reminiscent of the exclusive state in conventional MESI-like protocols. Therefore, a Commit instruction can complete even when the address is cached in the Dirty state, and a Reconcile instruction can complete even when the address is cached in the Clean state.

Each micro-protocol of Cachet embodies some *voluntary rules* that are not triggered by any specific instruction or protocol message. For example, at any time, a cache engine can write a dirty copy back to the memory or purge a clean copy from the cache. The memory engine can voluntarily send data to caches, provided that the memory contains the valid data. The existence of voluntary rules provides enormous scope for adaptivity which can be exploited to achieve better performance.

It is also possible to adaptively switch the micro-protocol that is operating on an address. For example, since the memory maintains no information about cache copies in Cachet-Base states, it can simply instruct a cache cell to adopt the Cachet-Base protocol when it does not have enough directory space to record the information. This can be important for large DSM systems when fully-mapped directory schemes are too expensive.

Since Cachet implements CRF, it is by definition a protocol for all high-level models whose programs can be translated into CRF programs. The translation can be performed statically by the compiler, or dynamically by the processor or the protocol engine. Thus, different high-level memory models can be used in different regions of memory simultaneously. For example, in an RC program, the region of memory used for input/output operations can have the SC semantics by simply employing an SC translation scheme for that region.

With both CRF and Cachet specified in TRS's, it can be proved formally that Cachet is a correct implementation of the CRF memory model [24]. The proof is based on simulation with respect to a mapping function that maps each Cachet term to a CRF term. The mapping function is defined in terms of the "drained states" which can be reached by applying a subset of Cachet rules. The simulation theorem shows that if a term t_1 can be rewritten to another term t_2 in Cachet, then the term corresponding to t_1 can be rewritten to the term corresponding to t_2 in CRF.

7. CRF Implications for Microarchitectures

First note that the CRF model can be implemented on any multiprocessor systems based on current microprocessors via a simple translation. For example, CRF programs can be executed correctly on a machine with Sparc's RMO model: Loadl and Storel are translated as normal Load and Store, Commit and Reconcile as Nop (no-operation), and Fence as Membar. It is safe to treat a Commit as a Nop because if it follows a Storel then its semantic effect is captured by the corresponding Store, otherwise it has no semantic effect. Similar argument applies to Reconcile.

In this section we discuss several issues that may arise if we were to develop a microprocessor to implement CRF directly. Since ordinary load and store instructions have been decomposed into finer-grain instructions in CRF, the instruction bandwidth needed to support a certain level of performance is likely to be high. This effect is similar to what was observed during the shift from CISC to RISC ISA's. A solution to this problem is discussed in Section 7.1.

Another important issue is the flexibility needed to dispatch CRF instructions to the memory system. The memory system, including the caches and associated protocols, works on the requests in the processor-to-memory queue. The processor must dispatch these requests quickly for good performance. This issue is discussed in Section 7.2.

Finally, as an aside, we discuss the impact of speculative load mechanism on CRF because it is a way of keeping load reordering transparent.

7.1. Reducing Instruction Bandwidth Requirement

While coarse-grain instructions alleviate some of the instruction bandwidth requirements, better encoding of load and store instructions using the CRF-bits can make a dramatic difference.

There are six CRF-bits: the Com and Rec bits are used to insert Commit and Reconcile operations, while the PreR, PreW, PostR and PostW bits are used to insert memory fences. Informally, the fence bits have the following effect if turned on:

- PreR: all preceding Loadl operations must complete before the instruction is performed;
- PreW: all preceding Commit operations must complete before the instruction is performed;
- PostR: the instruction must complete before any following Reconcile operation is performed;
- PostW: the instruction must complete before any following Storel operation is performed.

The following instruction sequences give the semantics of Load/Store instructions when all the CRF-bits are set:

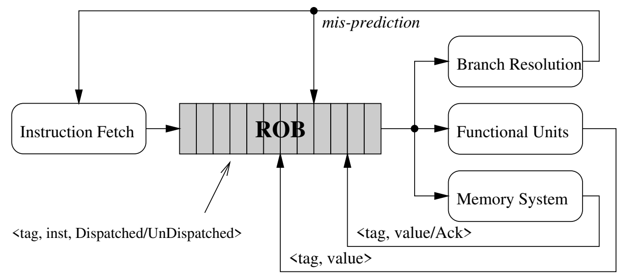


Figure 7. Instruction Dispatch from ROB

Load(a) [Rec,PreR,PreW,PostR,PostW]	Store(a,v) [Com,PreR,PreW,PostR,PostW]
Fence_{rr}(*,a);	Fence_{rw}(*,a);
Fence_{wr}(*,a);	Fence_{ww}(*,a);
Reconcile(a);	Storel(a,v);
Loadl(a);	Commit(a);
Fence_{rr}(a,*);	Fence_{wr}(a,*);
Fence_{rw}(a,*);	Fence_{ww}(a,*);

Notice, for a Load instruction it makes little sense to set the PreR or PreW bit without setting the the Rec bit. Similarly, the Com bit of a Store instruction should be set if the PostR or PostW bit is set. The Com and Rec bits are separate since synchronization instructions such as Test-&-Set and Swap behave as both a load and a store operation.

7.2. Dispatching the CRF Instructions

A modern processor fetches and decodes instructions and puts them in a reorder buffer (ROB) after register renaming (see Figure 7). Any instruction except a memory instruction can be dispatched to a functional unit as soon as its operands become available. Functional units store results back in the ROB. When a branch gets resolved, it can have the effect of killing all the instructions that are fetched after the branch, and resetting the program counter to the correct value. Instructions are retired from the ROB in the same order in which they are enqueued into the ROB. In such a system the order in which instructions retire from the ROB is the "program order" (further details of out-of-order and speculative execution can be found elsewhere [20, 22]).

Care has to be exercised in dispatching memory instructions from the ROB to the memory system (*i.e.*, pmb). If we treat the memory as a separate autonomous subsystem, then no speculative store should be dispatched because there is no way to retract or undo the effect of a store operation once it is dispatched. In addition, no memory instruction with an unresolved address or value should be dispatched. The rest of the constraints in dispatching of memory related instructions are determined by the memory model.

In SC, memory instructions are dispatched in the same order in which they reside in the ROB. In CRF, memory instructions can be dispatched earlier as long as the reordering rules are respected. For example, a Loadl(a) can be dispatched if there is no undispatched Storel or Reconcile in front that refers to address a or some unresolved address. Note that the dispatch can happen even if there is an undispatched $\mathsf{Fence_{rr}}$ or $\mathsf{Fence_{wr}}$ instruction in front whose post-address is a. The reason this is safe is that even if we dispatch instructions in-order, the reordering rules for CRF allow the instructions to be reordered in pmb. Thus, reordering at the processor-level cannot violate the memory model.

It should also be noted that the processor can simplify the memory interface by being less aggressive. For example, the processor may never dispatch a fence, and allow at most one outstanding load or store instruction for the same address. This will allow the memory system to reorder loads and stores as it pleases.

7.3. Impact of Speculative Loads

Some microprocessors such as MIPS R10000 have incorporated the capability for speculative execution of load instructions, which allows much more efficient implementation of SC [27]. A load instruction is allowed to be dispatched and performed speculatively before memory instructions preceding it have completed. However, all load and store instructions must be retired in-order. When an address is modified, a kill signal is issued to all the processors (this is a normal operation in systems with snoopy buses). When a processor receives the kill signal, it searches for load instructions to that address in its ROB which have obtained a value. If such a load is found, the load and all the following instructions are killed. This capability is easy to incorporate in any modern processor that does instruction reordering and speculative execution based on branch prediction.

It is worth pointing out that the speculative execution of load instructions requires the communication from the memory to a cache to be FIFO. Specifically, if the memory issues a kill signal to a cache, and then supplies a value to the same cache (probably for another address), then the kill signal must arrive at the cache first. This FIFO property is easily satisfied in an SMP because of the bus. However, in DSM systems where the memory is distributed among multiple sites, maintaining the FIFO order is difficult and expensive. Thus, even with speculative loads, SC may have limited scalability.

Finally we note that speculative loads would be an equally useful mechanism to implement the CR model, which is the same as the CRF model without instruction reordering.

8. Conclusion

We have proposed CRF, a mechanism-oriented memory model that provides great flexibility in both instruction reordering and data replication. Instruction reordering is constrained only by data dependences and memory fences. Data replication is facilitated by decomposing the load and store instructions into simpler instructions that operate on semantic caches and memory. Generally speaking, a Storel followed by a Commit forces the memory to be updated, and a Loadl preceded by a Reconcile retrieves the latest value from the memory. These fine-grain primitives provide architects and protocol designers more implementation flexibility in hiding long latency operations, especially in DSM's.

CRF is designed to serve as the interface between the compiler writer and the architect. It can be used as the common target machine language for compilers of high-level parallel languages. CRF is defined completely with only eight rewriting rules and a reordering table. All its mechanisms, notably, Commit, Reconcile and Fence, have direct instruction-level interpretation. CRF has precise semantics for any program, regardless of whether it is properly synchronized or not. This, we think, is essential for an architecture-centric memory model because of its impact on the ISA specification.

The complexity of weaker memory models, surprisingly, does not manifest itself in implementations but rather in a conceptual burden for the programmer and the architect. Indeed, the definitions of weaker models are not for the faint hearted but it is their unstability that is more problematic. No compiler writer or architect can deal with a memory model that changes with every computer generation even from the same manufacturer. CRF addresses both these issues squarely. Its definition is precise and small, and it provides a simple method of dealing with its variants. It essentially provides a systematic way for architectures and compiler implementations to evolve.

There is no broad consensus on the memory model future shared-memory machines should support. Hill has recently argued that multiprocessors should just support SC, or a model that just relaxes the ordering from writes to reads [13]. Hill's central argument is that the performance gap between SC and relaxed memory models can be narrowed by speculative execution and prefetching techniques [11] and thus, the complexity of weaker memory models is not justified. We agree with many of his observations but not his prescription.

For the sake of argument we compare SC with CR (CRF sans instruction reordering). As we pointed out earlier, the speculative load mechanism would be an equally useful mechanism to implement the CR model. However, CR provides more implementation flexibility. For example, CR allows a store operation to be performed without the exclusive ownership. This can be very useful to alleviate cache thrashing due to false-sharing. Different processors can work on different parts of the same cache line without interfering each other. Moreover, write-buffers can be employed so that store accesses to the same cache line can be merged to take advantage of burst bus transactions. Thus, even if the instruction reordering advantage turns out to be minimal, we expect Commit and Reconcile type mechanisms to widen the performance gap in future, especially for DSM's.

The next step in this research is to do a performance evaluation of the CRF model.

Acknowledgment: We gratefully acknowledge several general discussions about memory models with G. Gao, V. Sarkar and the member of the Cilk team, especially Mateo Frigo. Xiaowei Shen acknowledges the valuable summer experience in Charles Schulz's group at IBM T. J. Watson, especially in connection with the work on CACHET.

References

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

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

[3] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, June 1993.

[4] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. StarT-Voyager Parallel System. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), Paris, France*, pages 185–194, Oct. 1998.

[5] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, June 1996.

[6] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, Puerto Vallarta, Mexico, June 1998.

[7] G. R. Gao and V. Sarkar. Location Consistency – Stepping Beyond the Barriers of Memory Coherence and Serializability. Technical Memo 78, ACAPS Laboratory, School of Computer Science, McGill University, Dec. 1993.

[8] G. R. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Coherence Protocol. Technical Memo 16, CAPSL Laboratory, Department of Electrical and Computer Engineering, University of Delaware, Feb. 1998.

[9] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, Stanford University, 1995.

[10] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. In *Journal of Parallel and Distributed Computing*, pages 399–407, Aug. 1992.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, Aug. 1991.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. 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.

[13] M. D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, Aug. 1998.

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

[15] P. Keleher, A. 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.

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

[17] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[18] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. Morgan Kaufmann, 1994.

[19] R. S. Nikhil and Arvind. *Programming in pH – A Parallel Dialect of Haskell*. MIT, 1998.

[20] D. A. Patterson and J. L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, California, second edition, 1995.

[21] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th International Symposium On Computer Architecture*, pages 234–243, June 1987.

[22] X. Shen and Arvind. Processor Models. CSG Memo 400, Laboratory for Computer Science, MIT, June 1997.

[23] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, Laboratory for Computer Science, MIT, June 1997.

[24] X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. CSG Memo 414, Laboratory for Computer Science, MIT, Aug. 1998.

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

[26] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual (Version 9*. Prentice-Hall, 1994.

[27] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, Apr. 1996.