Hiding Relaxed Memory Consistency with Compilers

Jaejin Lee
Department of Computer Science & Engineering
Michigan State University
East Lansing, MI 48824
jlee@cse.msu.edu

David A. Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
padua@cs.uiuc.edu

Abstract

We present a compiler technique, which is based on Shasha and Snir's delay set analysis, to hide the underlying relaxed memory consistency model for an optimizing compiler for explicitly parallel programs. The compiler presents programmers with a sequentially consistent view of the underlying machine irrespective of whether it follows a sequentially consistent model or a relaxed model. To hide the underlying relaxed memory consistency model and to guarantee sequential consistency, our algorithm inserts fence instructions by identifying memory-barrier nodes. We reduce the number of fence instructions by exploiting the ordering constraints of the underlying memory consistency model and the property of fence and synchronization operations. We introduce dominators with respect to a node in a control flow graph to identify memory-barrier nodes. We also show that minimizing the number of memory-barrier nodes by using dominators with respect to a node is NP-hard.

1. Introduction

When users program a shared memory multiprocessor, they expect the behavior of the memory to be similar to that of a uniprocessor undertaking concurrent execution of several tasks. For example, consider a code that does busy-wait synchronization in Figure 1(A) in which threads are created at `cobegin` and threads are merged at `coend`. Most programmers would expect the output of the code in Figure 1(A) to be `flag=1, g=1, h=1`. However, under a memory architecture that aggressively relaxes orders between read and write accesses, after executing the code in Figure 1(A), we could get the result `flag=1, g=1, h=0`. The same result would be obtained by restructuring the code in Figure 1(A) to the code in Figure 1(B), where statements `g=1` and `flag=1` are reordered. The hardware reordering in the above example can be avoided by inserting fence instructions, as shown in Figure 1(C).

```
flag = 0
g = 0
cobegin
  g = 1
  flag = 1
||
  do while (flag==0)
  end do
  h = g
coend
print flag, g, h
```

(A)

```
flag = 0
g = 0
cobegin
  flag = 1
  g = 1
||
  do while (flag==0)
  end do
  h = g
coend
print flag, g, h
```

(B)

```
flag = 0
g = 0
cobegin
  g = 1
  fence
  flag = 1
||
  do while (flag==0)
  end do
  fence
  h = g
coend
print flag, g, h
```

(C)

Figure 1. **The effect of reordering instructions by the hardware.**

Lamport formalized an extension of the uniprocessor memory model for multiprocessors called sequential consistency [21]. Sequential consistency is arguably the most intuitive and natural memory consistency model for programmers [17]. Sequential consistency is what programmers assume when they program shared memory multiprocessors, even if they do not know exactly

what sequential consistency is.

Many shared memory multiprocessors follow a relaxed memory consistency model and provide a wide variety of hardware level optimizations, such as dynamic instruction reordering, speculative execution, and prefetching that take advantage of the memory model. There are many relaxed memory consistency models depending on the degree in which the order between read and write accesses is relaxed. Popular examples include processor consistency [15], weak ordering [3, 11], and release consistency [13]. There also are relaxed consistency models specific to processor architectures, such as DEC Alpha [40], IBM PowerPC [8], SUN SPARC [42], and Intel IA-64 [10]. The relaxed memory consistency model complicates programming and porting because the programmer is exposed to the various instruction reordering optimizations and the atomicity constraints of memory operations. In addition, variations in memory semantics must be considered when porting programs to guarantee correctness of execution. For example, the SUN SPARC V9 architecture supports two relaxed memory models (Partial Store Order (PSO) and Relaxed Memory Order (RMO)) as well as Total Store Order (TSO) model [42]. Switching the default memory model (TSO) into the conceptually more efficient PSO or RMO models would require redesigning the whole operating system running on SPARC multiprocessors [41]. The magnitude of the difficulties introduced by relaxed memory models have led some to argue that future systems should implement sequential consistency as their hardware memory consistency model because the performance boost of relaxed memory consistency models does not compensate for the burden placed on system software programmers [17, 14].

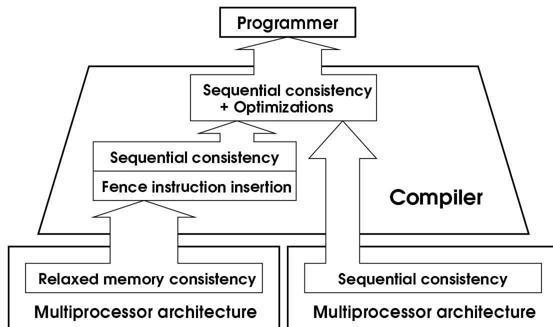


Figure 2. **The compiler.**

However, we do not believe it is necessary to go that far. If a compiler can control the underlying relaxed memory system by inserting fence instructions through sophisticated analysis, the programmer could treat the compiler and architecture as a sequentially consistent system and still profit from the performance advantage of the relaxed memory model.

In this paper, we describe techniques for an optimizing compiler for explicitly parallel programs. The characteristics of the compiler are [Figure 2]:

- the compiler provides programmers with a sequentially consistent view of the underlying architecture irrespective of the fact that it follows a sequentially consistent memory model or a relaxed memory model.

- the compiler makes it possible to apply compiler optimization techniques correctly to parallel programs that are not handled by conventional compilers.

We describe an algorithm based on Shasha and Snir's delay set analysis [38] to hide the underlying relaxed memory consistency model and to guarantee sequential consistency. Our algorithm inserts fence instructions by identifying memory-barrier nodes and reduces the number of memory-barrier nodes. We introduce the notion of *dominators with respect to a node* in a control flow graph to identify memory-barrier nodes. We also show that minimizing the number of memory-barrier nodes by using dominators with respect to a node is NP-hard. The analysis and optimization techniques for explicitly parallel programs are discussed in [20, 18, 35, 23, 24, 25, 26, 32, 33] and are beyond the scope of this paper.

The remainder of this paper is organized as follows. In the next section, we describe sequential consistency and delay set analysis. In section 3, we briefly describe the shared memory access ordering constraints of relaxed memory consistency models. In section 4, we describe the technique to hide the underlying relaxed memory model. We present the algorithm to guarantee sequential consistency under the weak ordering model and other relaxed memory consistency models. Section 5 presents related work and section 6 concludes our discussion.

2. Sequential Consistency and Delay Set Analysis

Simple uniprocessor memory semantics, in which a read to a variable returns the value most recently written to the variable, is what most programmers intuitively expect. Programmers assume that all memory operations in a program are executed in *program order*, where *program order* is the execution ordering of the operations executed by the source program. The

memory behavior of a uniprocessor undertaking the concurrent execution of several threads can be modeled by a total ordering that is constructed by merging the program ordering of all threads in such a way that the order within each thread is preserved in the total ordering. Sequential consistency [21] is a natural and intuitive multiprocessor extension of memory behavior of a uniprocessor undertaking the concurrent execution:

Definition 2.1 ***Sequential Consistency***
A multiprocessor system is sequentially consistent if the result of the execution of any program is the same as if all operations were executed in some global sequential order, and the operations of each parallel component appear in this sequence in the order specified by its program.

In this paper, a framework called *delay set analysis* is used to enforce sequential consistency. Delay set analysis was originally proposed by Shasha and Snir [38]. The analysis finds a minimal set of execution orderings that guarantees sequential consistency. Delay set analysis can be thought of as an extension of dependence analysis for explicitly parallel programs.

Let **P** be the order enforced by the source program (i.e., program ordering) between operations. Throughout this discussion, operations are assumed to be atomic. **P** is the *transitive closure* of the graph which contains the control flow edges of all the control flow graphs of each thread in the parallel program. *A node in the control flow graph represents an operation.* Two nodes m and n will be $m\mathbf{P}n$ if there is a path between m and n. Let **C** be a *conflict* relation on variable accesses. The *conflict* relation consists of the set of all pairs (v_i, v_j), where v_i and v_j are operations containing conflicting accesses. Two memory references *conflict* if they access the same memory location in different threads that might execute concurrently, and at least one of them is a write. This is a conservative definition of *conflict* relation compared to [38][1].

A delay relation **D** between two operations u and v forces v to wait until u completes execution. A *critical cycle* is a cycle of $\mathbf{P} \cup \mathbf{C}$ that has no chords [2] in **P**.

The delay relation **D** enforces sequential consistency if all **P** edges in the critical cycles appear in **D**. If **D** consists of all the **P** edges in the critical cycles, then **D** is a minimal delay relation that enforces sequential consistency in any execution of the program.

[1]Shasha and Snir in [38] state that "Two accesses conflict if the final value of the variable accessed, or the values computed by the accessing instructions may change when the order of accesses is reversed; two update accesses that commute, such as increment counter, do not conflict, even though both are writes."

[2]For two nonadjacent nodes u and v in a cycle, a chord is an edge (u, v).

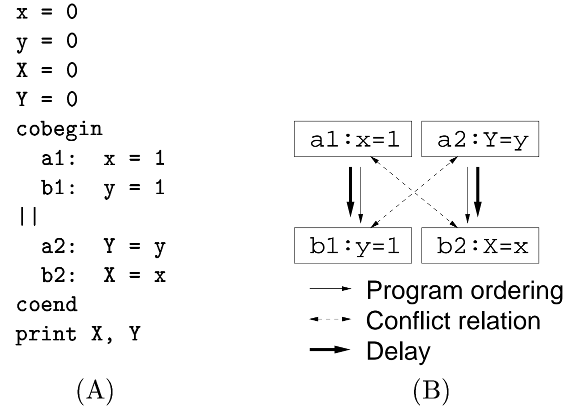


Figure 3. **An example of critical cycles and delays.**

For example, the code (presented in [38] and taken from Collier[7]) shown in Figure 3(A) gives an inconsistent outcome `X=0` and `Y=1` after the execution for the execution ordering `b1 a2 b2 a1`. There is a critical cycle (`b1, a2, b2, a1`) in Figure 3(B), and as a result, the **P** edges (`a1`, `b1`) $\in$ **P** and (`a2`, `b2`) $\in$ **P** (i.e., **D** = {(`a1`, `b1`), (`a2`, `b2`)}) should be enforced as delays to guarantee sequential consistency. In this example, we assume statements in the program are executed atomically.

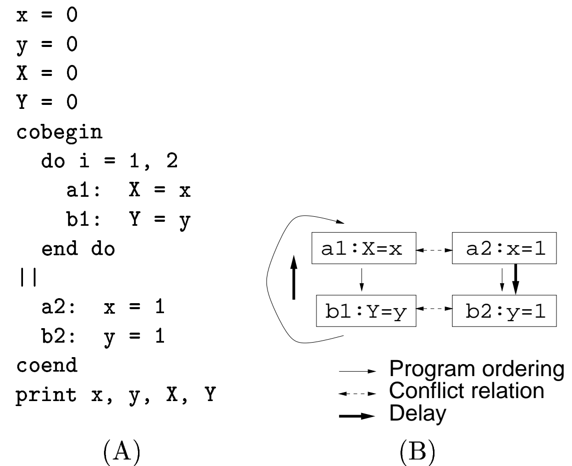


Figure 4. **An example of critical cycles and delays with a loop.**

One outcome of the code shown in Figure 4(A) is `x=1`, `y=1`, `X=0`, and `Y=1`. This is not sequentially consistent and is produced by the execution ordering `a1 a1 a2 b2 b1 b1` (there are many execution orderings that produce the outcome). There is a critical cycle (`a1, a2, b2, b1`) in Figure 4(B), and the **P** edges in

the cycle are (`b1`, `a1`) and (`a2`, `b2`). Thus, the minimal ordering $\mathbf{D} = \{$(`b1`, `a1`), (`a2`, `b2`)$\}$ must be enforced as delays. In other words, `b1` in iteration i must precede `a1` in iteration $i+1$, and `a2` and `b2` cannot be reordered.

3. Relaxed Memory Consistency Models

Relaxed memory consistency models reduce the restrictions on overlapping and reordering of memory operations, which are imposed by sequential consistency, in order to improve performance. Memory access operations to different memory locations can be reordered if reordering them does not violate explicit ordering constraints in relaxed memory consistency models. In this paper, we restrict our attention to three major relaxed memory consistency models: processor consistency [15, 13], weak ordering [11, 3], and release consistency [13].

shared			
competing unordered conflicting accesses			non-competing $\mathcal{R}_{\mathcal{NC}}$, $\mathcal{W}_{\mathcal{NC}}$
synchronization		non-synchronization $\mathcal{R}_{\mathcal{NS}}$, $\mathcal{W}_{\mathcal{NS}}$	
acquire $\mathcal{S}_{\mathcal{A}}$	release $\mathcal{S}_{\mathcal{R}}$		

Figure 5. **Categorization of shared memory accesses in release consistency model.**

Processor consistency is a weaker notion of sequential consistency but stronger than weak ordering. The ordering, a write followed by a read, that is required under sequential consistency is relaxed under the processor consistency model. The weak ordering model maintains program ordering only at the synchronization operations in the program. Release consistency is a combination of weak ordering and processor consistency (or sequential consistency). It further divides the synchronization operations into acquire and release accesses. As shown in Figure 5, release consistency divides shared memory operations into several categories. In weak ordering and release consistency, each memory access in a program is labeled according to this category. This information is exploited by the machine to enforce the ordering required by the consistency model.

It is easy to understand a memory model if we represent the model with orderings between reads and writes performed by a single processor [36]. Let $\mathcal{U}$ and $\mathcal{V}$ be two memory operations. $\mathcal{U} \rightarrow \mathcal{V}$ means that $\mathcal{V}$ executes after $\mathcal{U}$ has completed execution. We assume that a write is made visible to all other processors at the same time by a hardware cache coherence protocol if the multiprocessor system caches shared variables (i.e., the system satisfies write atomicity [2]).

The ordering constraints of each memory consistency model in a single processor are summarized in Figure 6. If there is a dependence between two memory operations, the program ordering between the two memory operations must be kept irrespective of the ordering constraints.

Model	Ordering constraints
Processor consistency	$\mathcal{R} \rightarrow \mathcal{R}$, $\mathcal{R} \rightarrow \mathcal{W}$, $\mathcal{W} \rightarrow \mathcal{W}$, $\mathcal{S} \rightarrow \mathcal{R}$, $\mathcal{S} \rightarrow \mathcal{W}$, $\mathcal{R} \rightarrow \mathcal{S}$, $\mathcal{W} \rightarrow \mathcal{S}$, $\mathcal{S} \rightarrow \mathcal{S}$
Weak ordering	$\mathcal{S} \rightarrow \mathcal{R}$, $\mathcal{S} \rightarrow \mathcal{W}$, $\mathcal{R} \rightarrow \mathcal{S}$, $\mathcal{W} \rightarrow \mathcal{S}$, $\mathcal{S} \rightarrow \mathcal{S}$
Release consistency (RC_{pc})	$\mathcal{S}_{\mathcal{A}} \rightarrow \mathcal{R}$, $\mathcal{S}_{\mathcal{A}} \rightarrow \mathcal{W}$, $\mathcal{R} \rightarrow \mathcal{S}_{\mathcal{R}}$, $\mathcal{W} \rightarrow \mathcal{S}_{\mathcal{R}}$, $\mathcal{R}_{\mathcal{NS}} \rightarrow \mathcal{R}_{\mathcal{NS}}$, $\mathcal{R}_{\mathcal{NS}} \rightarrow \mathcal{W}_{\mathcal{NS}}$, $\mathcal{W}_{\mathcal{NS}} \rightarrow \mathcal{W}_{\mathcal{NS}}$, $\mathcal{S}_{\mathcal{A}} \rightarrow \mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}} \rightarrow \mathcal{S}_{\mathcal{R}}$, $\mathcal{S}_{\mathcal{R}} \rightarrow \mathcal{S}_{\mathcal{R}}$, $\mathcal{R}_{\mathcal{NS}} \rightarrow \mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{R}} \rightarrow \mathcal{W}_{\mathcal{NS}}$

$\mathcal{R}$: read operation
$\mathcal{W}$: write operation
$\mathcal{S}$: synchronization operation
$\mathcal{S}_{\mathcal{A}}$: acquire operation
$\mathcal{S}_{\mathcal{R}}$: release operation
$\mathcal{R}_{\mathcal{NS}}$: non-synchronization read operation
$\mathcal{W}_{\mathcal{NS}}$: non-synchronization write operation
$\mathcal{R}_{\mathcal{NC}}$: non-competing read operation
$\mathcal{W}_{\mathcal{NC}}$: non-competing write operation
$\mathcal{R} = \mathcal{R}_{\mathcal{NS}} \cup \mathcal{R}_{\mathcal{NC}}$ and $\mathcal{W} = \mathcal{W}_{\mathcal{NS}} \cup \mathcal{W}_{\mathcal{NC}}$

Figure 6. **Ordering constraints given by each memory consistency model.**

4. Hiding Relaxed Memory Consistency with Compilers

In this section, we will explain how to exploit the ordering constraints of each relaxed memory consistency model to implement delays and how to reduce the number of delays and fence instructions.

4.1. Preserving Sequential Consistency with Delays

Suppose that $u\mathbf{D}v$, $v\mathbf{D}w$, and $u\mathbf{D}w$. It is not necessary to enforce $u\mathbf{D}w$ because it is implied by $u\mathbf{D}v$ and $v\mathbf{D}w$. In general, for a given $\mathbf{D}$, it is sufficient to enforce the *transitive reduction* of $\mathbf{D}$ [38]. A *transitive reduction* of $\mathbf{D}$, denoted by $\mathbf{D^{tr}}$, is the minimum relation such that $(\mathbf{D^{tr}})^+ = \mathbf{D}^+$, where $^+$ denotes the transitive closure relation [4].

Since each memory consistency model has its own ordering constraints, we do not need to enforce the delays that can be enforced by the ordering constraints. Let $\mathbf{D}$ be the delays found by the delay set analysis and $\mathbf{D_o}$ be the delays enforced by the ordering constraints (i.e., if $u\mathbf{D_o}v$, then $u \rightarrow v$ match an ordering constraint pattern in Figure 6). We want to find a minimal delay relation that enforces correctness together with $\mathbf{D_o}$. Then,

$$\mathbf{D_m} = ((\mathbf{D} \cup \mathbf{D_o})^+)^{\mathbf{tr}} - \mathbf{D_o}$$

is the minimal delay relation [38]. Thus, only the delays in $\mathbf{D_m}$ need to be implemented with special instructions, such as fences [13, 27, 36, 38] and synchronization operations depending on the consistency model.

The fence instructions of commercial architectures have various names and semantics: store barrier in the SPARC V8 architecture; read-read, read-write, write-read, and write-write fences (MEMBAR) in the SPARC V9 architecture; memory barrier and write memory barrier (MB) in Alpha; SYNC in MIPS and the PowerPC architecture; and, memory fence (mf) in the Intel IA-64 architecture. Because the semantics of a fence differ from architecture to architecture, we assume that a *fence* has the following semantics:

Definition 4.1 ***Fence***
A fence instruction imposes ordering between memory operations in such a way that when a fence instruction is executed by a processor, all previous memory operations of the processor are guaranteed to have completed. Furthermore, no memory operation of the processor that follow the fence instruction in the program is issued until the fence completes execution.

4.2. Exploiting the Property of Fence and Synchronization Instructions

Fences are inserted at a node of the control flow graph to enforce one or more delays. Fences can be added as separate instructions or, if the node is a shared memory operation, the operation can be replaced by a special load or store instruction that contains fine-grain fences, such as PreFenceW and PostFenceR in the CRF model [39].

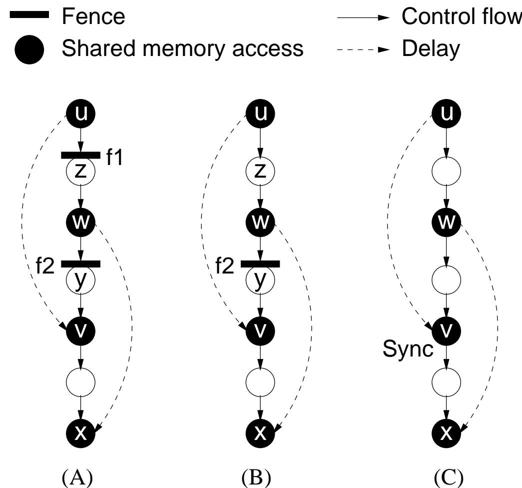


Figure 7. **Inserting fence instructions.**

A naive algorithm may insert more fences than needed to enforce sequential consistency. For example, consider the delays $u\mathbf{D_m}v$ and $w\mathbf{D_m}x$ in Figure 7(A). These delays are in the same thread. Two fences, *f1* and *f2*, are inserted at nodes z and y to enforce the delays. If y executes whenever u and v executes, then we do not need fence *f1* at z [Figure 7(B)]. If we find such y, the fence used to enforce the delay $w\mathbf{D_m}x$ can be used to enforce the delay $u\mathbf{D_m}v$.

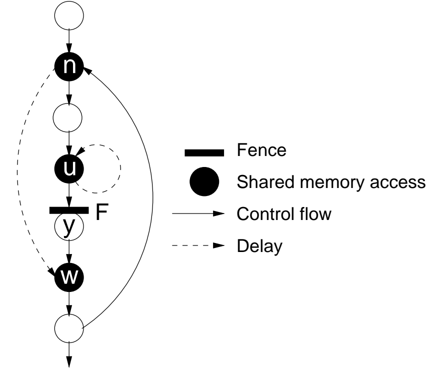


Figure 8. **The fence F is enough to enforce both of the delays $n\mathbf{D}w$ and $u\mathbf{D}u$.**

In the case of weak ordering, we can further reduce the number of fences by exploiting the property of synchronization operations. Consider the delays $u\mathbf{D_o}v$ and $w\mathbf{D_m}x$ in Figure 7(C). All the ordering constraints in the weak ordering model are given with respect to synchronization operations $\mathcal{S}$ [Figure 6]:

$$\mathcal{S} \rightarrow \mathcal{R},\ \mathcal{S} \rightarrow \mathcal{W},\ \mathcal{R} \rightarrow \mathcal{S},\ \mathcal{W} \rightarrow \mathcal{S},\ \mathcal{S} \rightarrow \mathcal{S}$$

The delays that match the patterns of the ordering constraints belong to $\mathbf{D_o}$. These delays contain explicit synchronization operations in the source program. Operation v is identified as an explicit synchronization variable access and is labeled *Sync* in Figure 7(C). If v executes whenever w and x executes, then the delay $w\mathbf{D_m}x$ (note that this delay does not belong to $\mathbf{D_o}$ because $w \rightarrow x$ does not match any pattern of the constraints) is enforced by the ordering constraints $u \rightarrow v$ and $v \rightarrow x$ which are enforced by the weak ordering model because v is a synchronization operation. Thus, we do not need to insert any fences or (artificially) mark the nodes w and x as a synchronization operation to enforce the delay $w\mathbf{D_m}x$ in Figure 7(C).

4.3. Identifying Memory-Barrier Nodes

In this section, we explain how to identify synchronization operations in weak ordering. We will use the solution for weak ordering to deal with other relaxed memory consistency models. In weak ordering,

we identify a node y as a synchronization instruction to enforce a delay $u\mathbf{D_m}v$ if y always executes after u and before v whenever u and v execute. We assume that a special bit inserted in each instruction is used to mark the instruction as a synchronization operation. If the bit is set, the machine treats the instruction as a synchronization operation. If a node contains an instruction with the special bit set, we call the node a *memory-barrier node* from now on. A conservative condition for finding a memory-barrier node y that enforces a delay $u\mathbf{D_m}v$ is: *If every path from u to v in the control flow graph of a thread goes through y, then y executes whenever u and v execute.* We want to find a memory-barrier node that enforces as many delays as possible. The example shown in Figure 8 describes this situation. Fence F is enough to enforce both delays $n\mathbf{D_m}w$ and $u\mathbf{D_m}u$. To detect this case, we need to examine all the paths from u to u and n to w, and check whether y is common in these paths.

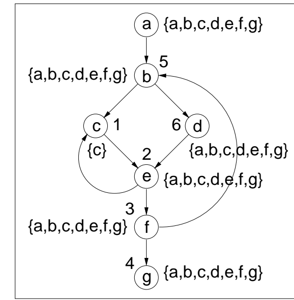
To find memory-barrier nodes, we introduce the notion of *dominators with respect to a node.* A node n *dominates* a node m (n **dom** m) if every control flow path from the program entry node to m goes through n. A node n *postdominates* a node m (n **pdom** m) if every control flow path from m to the program exit node goes through n. The (classical) dominators of a node m are the *dominators with respect to the program entry node* of the control flow graph.

Definition 4.2 ***Dominators w.r.t. a Node***
A node s dominates a node v with respect to a node u if every control flow path from u to v goes through s. This relation is denoted by $s\ \mathbf{dom_u}\ v$.

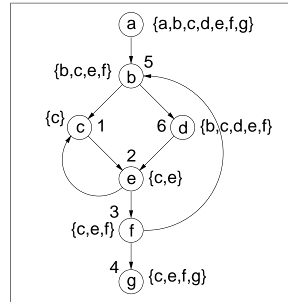
If a node v is unreachable from u, then v has an empty set of dominators with respect to u. The unreachable nodes from u can be found by depth-first traversal of the control flow graph with the node u as a root node. If a node d dominates each predecessor of a node $n \neq d$ with respect to a node u, then d must dominate n with respect to u. We use an iterative data flow framework to compute dominators with respect to a node. The algorithm traverses the graph in depth-first manner. Let $\mathbf{dom_u}[n]$ be the set of nodes that dominate n with respect to u, and let $\mathbf{pred}[n]$ be the set of predecessors of n in the control flow graph. Let U be the set of nodes that are unreachable from u. Then, the data flow equation is:

$$\mathbf{dom_u}[n] = \{n\} \cup (\bigcap_{p \in \mathbf{pred}[n]} \mathbf{dom_u}[p]),$$
$$\text{where } n \neq u \text{ and } n \notin U.$$

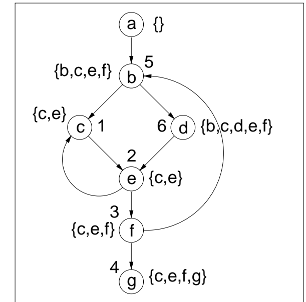
We use the iterative algorithm for classical dominators [6, 5] to find dominators with respect to a node u



(A)



(B)



(C)

Figure 9. **An example of finding dominators with respect to a node c.**

by treating u as the program entry node. At the beginning, each set $\mathbf{dom_u}[n]$ for a node $n \neq u$ is initialized with N, the set of all nodes, and the set $\mathbf{dom_u}[u]$ is initialized with $\{u\}$.

After we have reached the fixed point of the algorithm, we update $\mathbf{dom_u}[u]$ with $\{u\} \cup (\bigcap_{p \in \mathbf{pred}[u] \wedge p \notin U} \mathbf{dom_u}[p])$ and $\mathbf{dom_u}[n]$ of $n \in U$ with $\emptyset$.

For example, consider the flow graph in Figure 9(A). We want to find dominators with respect to the node c. The set of dominators with respect to c, $\mathbf{dom_c}[n]$, for each node n is initialized with {a,b,c,d,e,f,g}, except for the node c whose set is initialized with {c}. The number next to a node is the depth-first search (*dfs*) number for the node from the root node c. The contents of $\mathbf{dom_c}[n]$ after the first iteration for each node n is in Figure 9(B). The node a is unreachable from c. Note that we do not change the dominator sets of c nor the unreachable nodes during the iteration. In Figure 9(C), after we reach the fixed point of the iteration, we update $\mathbf{dom_c}[c]$ with $\mathbf{dom_c}[b] \cap \mathbf{dom_c}[e]$. Then, we update $\mathbf{dom_c}[a]$ of the unreachable node a with $\emptyset$. All the dominators with respect to each node in the flow graph is shown in Figure 10.

For an iterative data flow algorithm that uses depth-first ordering, the upper bound on the number of passes

nodes\with respect to	a	b	c
a	a		
b	a,b	b,e,f	b,c,e,f
c	a,b,c	b,c	c,e
d	a,b,d	b,d	b,c,d,e,f
e	a,b,e	b,e	c,e
f	a,b,e,f	b,e,f	c,e,f
g	a,b,e,f,g	b,e,f,g	c,e,f,g

nodes\with respect to	d	e	f	g
a				
b	b,d,e,f	b,e,f	b,f	
c	c,d,e	c,e	b,c,f	
d	b,d,e,f	b,d,e,f	b,d,f	
e	d,e	e	b,e,f	
f	d,e,f	e,f	b,e,f	
g	d,e,f,g	e,f,g	f,g	g

Figure 10. **The dominators with respect to a node in the graph in Figure 9.**

taken by the algorithm is two plus the *depth* of the flow graph [5, 34]. Because there are $O(|E|)$ bit vector operations in each pass and each pass traverses $O(|N|)$ nodes, the time complexity of the algorithm is $O((d+2) \times (|N|+|E|))$, where d is the depth of the control flow graph.

4.4. Minimizing the Number of Memory-Barrier Nodes by Using Dominators with Respect to a Node is NP-hard

In this section, we show that minimizing the number of memory-barrier nodes is an NP-hard problem by proving that its decision version is an NP-complete problem. We call the decision problem **MIN NODES**. An instance $(G, \mathbf{D_m}, k)$ of **MIN NODES** consists of a directed graph G, a finite set $\mathbf{D_m}$ of delays, and a positive integer k. The problem is to find a set $S \subseteq \bigcup_{(u,v)\in \mathbf{D_m}} \mathbf{dom_u}[v]$ of size k whose members enforce all the delays in $\mathbf{D_m}$. A node n enforces a delay $u\mathbf{D_m}v$ if $n \in \mathbf{dom_u}[v]$.

Lemma 4.1 **MIN NODES** *belongs to the class NP.*

Proof: To show that **MIN NODES** $\in$ NP, we check whether the members of the set $S \subseteq \bigcup_{(u,v)\in \mathbf{D_m}} \mathbf{dom_u}[v]$ enforce all the delays in $\mathbf{D_m}$ and whether S has size k. Checking whether S has size k can be done in linear time by counting its members. In order to check whether a node $n \in S$ enforces a delay $u\mathbf{D_m}v$, we first check if there is a path from u to v that goes through n. Then, we construct a subgraph G' of G where the node n and its incoming and outgoing edges are removed. If there is a path from u to v that goes through n in G and if there is no path from u to v in G', then the node n belongs to $\mathbf{dom_u}[v]$ and enforces the delay $u\mathbf{D_m}v$. This check can be done in polynomial time. □

Lemma 4.2 **MIN NODES** *is NP-hard.*

Proof: To show that the **MIN NODES** problem is NP-hard, we define a polynomial time reduction function that maps every instance of the **VERTEX COVER** [12] problem to an instance of the **MIN NODES** problem. An instance (G_{VC}, k) of the **VERTEX COVER** problem consists of an undirected graph $G_{VC} = (V_{VC}, E_{VC})$ and a positive integer k. The problem is finding a subset $V' \subseteq V_{VC}$ of size k such that for each edge $(u, v) \in E_{VC}$ at least one of u and v belongs to V'. A *vertex cover* of a undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that each edge of G is incident upon some vertex in S.

We identify each element $n \in V_{VC}$ by a unique positive integer in $\{1, \ldots, |V_{VC}|\}$. The **VERTEX COVER** problem can be reduced into the **MIN NODES** problem by defining the graph G and the delays $\mathbf{D_m}$:

- $G = (V, E)$, where

$$V = V_{VC} \cup \{entry, exit\}$$
$$E = \{(u,v) | \{u,v\} \in E_{VC} \wedge u \leq v\} \cup \{(entry, 1), (|V_{VC}|, exit)\}$$

- $\mathbf{D_m} = \{(u,v) | \{u,v\} \in E_{VC} \wedge u \leq v\}$ (i.e., each edge in E_{VC} corresponds to a delay in $\mathbf{D_m}$).

Obviously, this can be done in polynomial time.

For example, we have an instance (G_{VC}, k) of the **VERTEX COVER** problem [Figure 11(A)], where $G_{VC} = (V_{VC}, E_{VC})$:

$$V_{VC} = \{1,2,3,4,5,6\}$$
$$E_{VC} = \{\{1,2\},\{2,3\},\{2,4\},\{3,5\},\{4,5\}\{5,6\}\}$$

Then, we have an instance $(G, \mathbf{D_m}, k)$ of the **MIN NODES** problem [Figure 11(B)], where $G = (V, E)$:

$$V = \{1,2,3,4,5,6, entry, exit\}$$
$$E = \{(1,2),(2,3),(2,4),(3,5),(4,5),(5,6), (entry, 1), (6, exit)\}$$
$$\mathbf{D_m} = \{(1,2),(2,3),(2,4),(3,5),(4,5),(5,6)\}$$

Now, we show that this transformation is indeed a reduction. First, we show that given a vertex cover S of size k, the set S enforces all the delays in $\mathbf{D_m}$. For each vertex $n \in S$, there exists at least one edge

$(u, v) \in E_{VC}$ (without loss of generality, we assume $u \leq v$) such that $u = n$ or $v = n$. In addition, the edge (u, v) corresponds to a delay $u\mathbf{D_m}v$. Because there is an edge (u, v) in G for each delay $u\mathbf{D_m}v$, $\mathbf{dom_u}[v] = \{u, v\}$. Therefore, $n \in \mathbf{dom_u}[v]$. The set S enforces all the delays in $\mathbf{D_m}$.

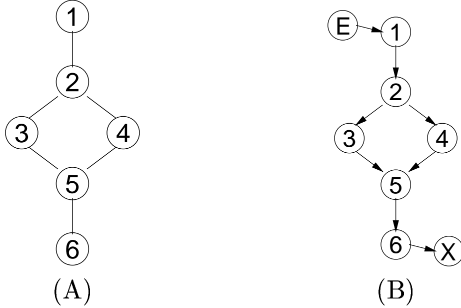


Figure 11. **An instance of the VERTEX COVER problem and the corresponding MIN NODES problem.**

Conversely, suppose that a set S of size k enforces all the delays in $\mathbf{D_m}$. Assume a node $n \in S$ enforces a delay $u\mathbf{D_m}v$ (i.e., $n \in \mathbf{dom_u}[v]$). Because there is an edge (u, v) in G for each delay $u\mathbf{D_m}v$, $\mathbf{dom_u}[v] = \{u, v\}$ (i.e., $n = u$ or $n = v$). Also, the delay $u\mathbf{D_m}v$ corresponds to the edge (u, v) in G_{VC}. Therefore, S is the vertex cover of size k. □

Theorem 4.1 MIN NODES *is NP-complete.*

Proof: By Lemma 4.1 and Lemma 4.2. □

A naive algorithm for minimizing the number of memory-barrier nodes will check each subset of the nodes in $\bigcup_{u\mathbf{D_m}v} \mathbf{dom_u}[v]$ to determine whether the nodes in the subset enforces all the delays $\mathbf{D_m}$. This takes time $O(2^{|\bigcup_{\text{u and v such that } u\mathbf{D_m}v} \mathbf{dom_u}[v]|} \times t)$, where $t =$ the time taken to check whether a subset enforces all the delays in $\mathbf{D_m}$. Instead, we use an approximation algorithm to minimize the number of memory-barrier nodes. This is a slight modification of the greedy approximation algorithm developed to solve the optimization version of the **MINIMUM COVER** problem [9]. An instance (X, C) of **MINIMUM COVER** consists of a finite set X and a collection C of subsets of X such that every element of X belongs to at least one subset in C. The problem is finding a minimum subset $S \subseteq C$ whose members cover all elements in X. A subset $S' \in C$ covers its elements.

The minimization algorithm is shown in Figure 12. It converts **MIN NODES** into **MINIMUM COVER**. Let $\mathbf{D_m}$ be the delays in **MIN NODES**. Then,

$$X = \mathbf{D_m}$$

$$C = \{C_n \mid C_n = \{(u, v) \mid (u, v) \in \mathbf{D_m} \wedge n \in \mathbf{dom_u}[v]\}\}$$

The set C_n in the **for** loop contains the delays that are enforced by a node $n \in \bigcup_{(u,v)\in\mathbf{D_m}} \mathbf{dom_u}[v]$. At each iteration of the **while** loop, X contains the remaining delays that have not been enforced. A set C_n containing as many unenforced delays in X as possible is chosen in each iteration of the **while** loop. If there is a tie between C_n sets, we choose a set that corresponds to the node that is located in a less frequently executed path in the control flow graph. Finally, the set M contains memory-barrier nodes that enforces all the delays in $\mathbf{D_m}$.

The outer **for** loop in Figure 12 takes $O(|U||\mathbf{D_m}|)$, where $U = \bigcup_{(u,v)\in\mathbf{D_m}} \mathbf{dom_u}[v]$. The **while** loop is executed at most $min(|X|, |C|) = min(|\mathbf{D_m}|, |U|)$ times, and the loop body is calculated in $O(|X||C|) = O(|\mathbf{D_m}||U|)$. Since $|\mathbf{D_m}| \leq |U|$, the time complexity of the algorithm is:

$$O(|U||\mathbf{D_m}|) + O(|\mathbf{D_m}||U|min(|\mathbf{D_m}|, |U|)) = O(|\mathbf{D_m}|^2|U|)$$

The solution found by the greedy algorithm for **MINIMUM COVER** is not more than $H(max|S| : S \in C)$ times as large as an optimal solution [9], where $H(n)$ is the nth harmonic number. Thus, the set M found by the algorithm in Figure 12 is not more than $H(max|S| : S \in C)$ times as large as an optimal set of memory-barrier nodes, where $C = \{C_n \mid C_n = \{(u, v) \mid (u, v) \in \mathbf{D_m} \wedge n \in \mathbf{dom_u}[v]\}\}$.

```
1  U ← ⋃_{(u,v)∈Dm} dom_u[v]
2  C ← ∅
3  for each n ∈ U
4    C_n ← ∅
5    for each uDmv
6      if n ∈ dom_u[v] then
7        C_n ← C_n ∪ {(u, v)}
8      end if
9    end for
10   C ← C ∪ {C_n}
11 end for
12 X ← Dm
13 M ← ∅
14 while X ≠ ∅
15   Select a C_n ∈ C that maximize | C_n ∩ X |.
16   X ← X − C_n
17   M ← M ∪ {n}
18 end while
```

Figure 12. **Minimizing the number of memory-barrier nodes.**

4.5. Profitability

We also need to consider profitability when we identify memory-barrier nodes and when we insert a fence. Since the goal of relaxed memory consistency model is to make memory operations to different locations overlapped (pipelined) or reordered in order to hide the memory latency, we want to insert a fence as close to node v as possible in order to maximize the reordering and overlapping by processors if (u,v) is a delay. Also, it is more desirable to insert a fence at the memory-barrier node $n \in \mathbf{dom_u}[v]$ that is located in a less frequently executed path.

For example, consider Case1 of Figure 13. Node v is more desirable than any other nodes in $\mathbf{dom_u}[v]$ (say that u is a write and v is a read) to maximize overlapping and reordering of memory operations. If there are multiple delays $u_1\mathbf{D_m}v_1$, ..., $u_n\mathbf{D_m}v_n$ and if $\bigcap_{1\leq i\leq n}\mathbf{dom_{u_i}}[v_i] \neq \emptyset$ and the nodes in $\bigcap_{1\leq i\leq n}\mathbf{dom_{u_i}}[v_i]$ are ordered by the dominators with respect to a node relation, we choose the latest node in $\bigcap_{1\leq i\leq n}\mathbf{dom_{u_i}}[v_i]$ as the memory-barrier node to enforce the delays. In Case2, inserting a fence at the node u, m, or $n \in \mathbf{dom_u}[v]$ is more desirable. In Case3, o, p, or v are more desirable. For Case4, o, p or v are the best choices because other nodes are in the loop. These schemes can be combined with the line 15 of the minimization algorithm in Figure 12 when there is a tie between nodes.

For Case5, the memory-barrier node will be a node in $\mathbf{dom_u}[v] \cap \mathbf{dom_m}[s] = \{m,n,q\}$. However, if we identify both s and v as memory-barrier nodes, it maximizes overlapping and reordering of memory operations. Even though the static number of memory-barrier nodes are increased in this case, the number of memory-barrier nodes executed is the same. If there is such a case, we can move the memory-barrier to s and v.

4.6. Limitations of the Algorithm Based on Dominators with Respect to a Node

There are some limitations of the algorithm based on dominators with respect to a node. Consider Figure 14 Case1. We have $\mathbf{dom_u}[v] = \{u,q,x,v\}$, $\mathbf{dom_s}[w] = \{s,t,w\}$, and $\mathbf{dom_o}[n] = \{o,p,n\}$. Suppose that the identified memory-barrier nodes are n, w, and v after applying the algorithm. However, the memory-barrier node v is redundant.

As another example, consider Figure 14 Case2. We have $\mathbf{dom_u}[v] = \{u,q,x,v\}$ and $\mathbf{dom_s}[w] = \{s,t,w\}$. Suppose that the identified memory-barrier nodes are

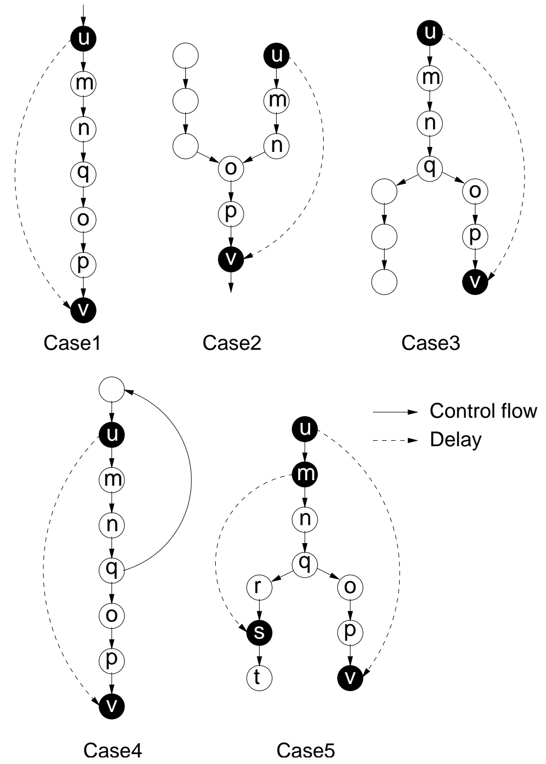


Figure 13. **Profitability of inserting a fence instruction for a delay.**

w and v after applying the algorithm. It is more desirable to make one of the nodes o, p, n, and m a memory-barrier node because every time this code is executed, only one synchronization operation will be executed to enforce both of the delays. However, to detect these cases in general, we must examine all the possible executable paths from u to v and check whether there is a memory-barrier node in each path.

4.7. Other Models

We can use the solution for weak ordering to deal with other memory consistency models. We insert a (coarse grain) fence instructions immediately before or immediately after each memory-barrier node found by the minimization algorithm in Figure 12.

When we apply the minimization algorithm in Figure 12 to other memory models, the memory-barrier node n in Figure 15 gets only one fence because the algorithm aimed at minimizing the number of *nodes*. The node n needs two fences to enforce the two delays [Figure 15 Case1 and Case2]. To prevent this, we split a node into two nodes before applying the minimization algorithm if the node is both source and sink of more than one delays and if the operation in the node

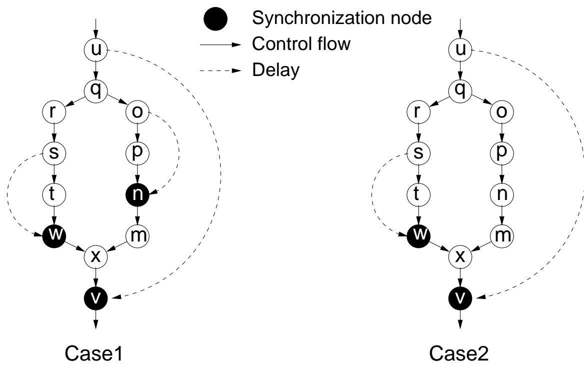


Figure 14. **Limitations of the algorithm based on dominators with respect to a node.**

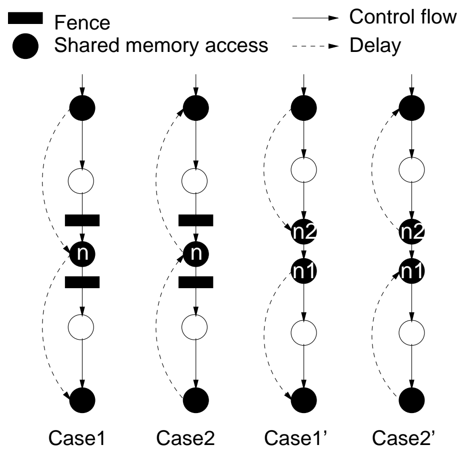


Figure 15. **Inserting fence instructions at a node n that is both source and sink of more than one delays.**

cannot be replaced with a special load or store instruction that contain fine grain fences [Figure 15 Case1′ and Case2′]. Enforcing delays in the weak ordering model is summarized in Figure 16. Enforcing delays in the processor consistency model and in the release consistency model are similar to enforcing delays in the weak ordering model.

5. Related Work

Despite the importance of the topic, there is no previous work that deals with compiler techniques to hide relaxed memory consistency models.

To overcome the difficulties of programming systems with the relaxed memory consistency model, Adve and Gharachorloo proposed a programmer-centric specification approach [13, 3, 1, 2]. It requires the programmer to provide certain correctness information about the program to the system. This information is used by

Let $\mathbf{D}$ be the delays found by the delay set analysis and $\mathbf{D_o}$ be the delays enforced by the ordering constraints.

$\mathbf{D_m} \leftarrow ((\mathbf{D} \cup \mathbf{D_o})^+)^{\mathbf{tr}} - \mathbf{D_o}$
for each $u\mathbf{D_m}v$
  Compute $\mathbf{dom_u}[v]$.
end for
Compute the set M of nodes using
  the algorithm in Figure 12.
Label each node in M with `Sync`.

Figure 16. **Enforcing delays in the weak ordering model.**

the system to avoid applying optimizations that violate the correctness (i.e., sequential consistency) of the program execution. However, it is hard for programmers to infer correctness of the program under the relaxed memory models.

Shen, Arvind, and Rudolph proposed Commit-Reconcile & Fences (CRF) memory consistency model [39]. The memory model is defined by using term rewriting system and gives clear algebraic semantics that is not provided by other memory consistency models. The major role of the CRF model is to serve as an interface between programmer-centric memory models and the real implementations of memory models. It provides compiler writers with a well defined interface to the underlying memory consistency model.

Java [16, 22] supports concurrent programming and its memory model is a relaxed memory model. It is not sequentially consistent and is very hard to understand. A study by Pugh [37] described some of the problems with the current Java memory model in the sense of compiler optimizations and coherence.

Krishnamurthy and Yelick [19, 20] developed a *back path* finding algorithm to find delays in SPMD programs. The time complexity of the back path finding algorithm is $O(n^3)$ for SPMD programs, where n is the number of remote memory access operations in the program. Midkiff, Padua, and Cytron [33] proposed a delay set analysis technique to deal with programs with loops and arrays. In their work, the conflict relations between statements are represented by a static conflict relation and a dynamic conflict relation, and a collection of Diophantine equations and inequalities are used to find critical cycles.

Li and Abu-sufah [28, 29], also Midkiff and Padua [30, 31], developed an algorithm to reduce the number of synchronized memory references to shared data in *doacross* loops. They identify the array references

whose synchronized accesses will eliminate the synchronized accesses of other array references. Their technique reduces the number of references that have to be synchronized due to true-dependence or anti-dependence in *doacross* loops.

6. Conclusions

We described a compiler technique that hides an underlying relaxed memory consistency model by using Shasha and Snir's delay set analysis so that the compiler presents to the programmer an intuitive and natural programming model based on sequential consistency. It shifts the programmer's burden of considering the underlying machine architecture to the compiler. This facilitates programming and debugging.

To guarantee sequential consistency by hiding the underlying relaxed memory consistency model, we identify a memory-barrier node for each delay found by the delay set analysis. We introduced dominance with respect to a node relation in order to locate memory-barrier nodes in the control flow graph. In addition, we showed that minimizing the number of memory-barrier nodes by using the dominance with respect to a node is NP-hard. An important implication of identifying memory-barrier nodes is that the compiler can freely apply instruction reordering techniques to the code section in between two consecutive memory-barrier nodes.

Studies of compilers and programming languages have not given much attention to the relationship between correctness and ease of programming issues with multiprocessor memory consistency models. We have shown in this paper one method to guarantee correctness (i.e., sequential consistency) and to facilitate programming, but there is still much to be done. For example, our algorithms do not deal with the case where it is not necessary to enforce a delay to guarantee sequential consistency. Indeed, enforcing delays is just a sufficient condition to guarantee sequential consistency.

7. Acknowledgements

This work is supported in part by an IBM Cooperative Fellowship, a Partnership Award from IBM, Michigan State University, NSF contract ACI 98-70687, and NSF contract DMS 98-73945. This work is not necessarily representative of the positions or policies of the Army, Government or IBM Corp.

The authors thank Samuel P. Midkiff for his comments on early drafts of this paper. We also thank the anonymous reviewers for their helpful comments.

References

[1] Sarita Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, University of Wisconsin-Madison, December 1993. Computer Science Technical Report #1198.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[3] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of The 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–14, May 1990.

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[6] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, New York, 1998.

[7] W. Collier. Principles of architecture for systems of parallel processes. Technical Report TR00.3100, IBM T. J. Watson Research Center, March 1981.

[8] Apple Computer, IBM, and Motorola. *PowerPC Microprocessor Common Hardware Reference Platform.* Morgan Kaufmann Publishers, Inc., 1995.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[10] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, May 1999. Rev. 1.0.

[11] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of The 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 434–442, June 1986.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of The 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–26, May 1990.

[14] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of The 26th Annual International Symposium on Computer Architecture (ISCA)*, pages 162–171, May 1999.

[15] James R. Goodman. Cache consistency and sequential consistency. Technical Report CS-TR-91-1006, Department of Computer Science, University of Wisconsin, February 1991.

[16] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[17] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[18] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analysis for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[19] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel SPMD programs. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[20] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–204, June 1995.

[21] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[22] Doug Lea. *Concurrent Programming in Java.* Addison Wesley, 1996.

[23] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1999. Department of Computer Science Technical Report UIUCDCS-R-99-2112.

[24] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 114–130. Springer, August 1997.

[25] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. A constant propagation algorithm for explicitly parallel programs. *International Journal of Parallel Programming*, 26(5):563–589, 1998.

[26] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of The 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1999.

[27] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann Publishers, Inc., 1995.

[28] Zhiyuan Li and Walid Abu-sufah. A technique for reducing synchronization overhead in large scale multiprocessors. In *Proceedings of The 12th Annual International Symposium on Computer Architecture (ISCA)*, pages 284–291, 1985.

[29] Zhiyuan Li and Walid Abu-sufah. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers*, C-36(1):105–109, January 1987.

[30] Samuel P. Midkiff and David A. Padua. Compiler generated synchronization for do loops. In *the 1986 International Conference on Parallel Processing*, pages 19–22, August 1986.

[31] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[32] Samuel. P. Midkiff and David. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing (ICPP), Vol. II Software*, pages 105–113, August 1990.

[33] Samuel P. Midkiff, David A. Padua, and Ron Cytron. Compiling programs with user parallelism. In *Languages and Compilers for Parallel Computing*, pages 402–422, 1990.

[34] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[35] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *Proceedings of the 1998 International Conference on Parallel Processing*, August 1998.

[36] David A. Patterson and John L. Hennessy. *Computer Architecture: A Qunatitative Approach.* Morgan Kaufmann Publishers, Inc., second edition, 1996.

[37] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[38] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[39] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A new memory model for architects and compiler writers. In *Proceedings of The 26th Annual International Symposium on Computer Architecture (ISCA)*, pages 150–161, May 1999.

[40] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual.* Digital Press, second edition, 1995.

[41] SUN Microsystems Technical Support, December 1998. Personal Communication.

[42] David L. Weaver and Tom Germond. *The SPARC Architecture Manual.* Prentice-Hall, 1994.