Hiding Relaxed Memory Consistency with a Compiler

Jaejin Lee and David A. Padua

*IEEE Transactions on Computers 50*, 8 (August 2001)

Adapted for presentation by Zvezdan Petković

October 5, 2001
Overview

➤ Introduction

➤ Sequential Consistency and Delay Set Analysis

➤ Relaxed Memory Consistency Models

➤ Hiding Relaxed Memory Consistency

✦ Preserving Sequential Consistency Using Delays
✦ Inserting Fences
✦ Marking Instructions as Synchronization Operations
✦ Identifying Memory-Barrier Nodes
- Minimizing the Number of Memory-Barrier Nodes
- Profitability
- Limitations of the Algorithm Based on Dominators with Respect to a Node
- Other Models

➤ Related Work

➤ Conclusions
Introduction

➤ most programmers expect shared-memory multiprocessors to behave like uniprocessors under a multiprogramming OS

➤ Lamport has formalized sequential consistency as an extension of the uniprocessor memory model for multiprocessors [1979]

➤ programmers of SMPs assume sequential consistency, even if they do not know exactly what it is

➤ however, many SMPs can reorder reads and writes, and provide various hardware level optimizations to take advantage of the relaxed memory consistency model
flag = 0
g = 0
cobegin
  g = 1
  flag = 1
||
do while (flag == 0)
end do
h = g
coenend
print flag, g, h

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

(a)

(b)

FIGURE 1: The effect of reordering instructions by the hardware.
relaxed memory consistency models can be classified

- according to the degree in which the order between read and write access is relaxed (e.g. processor, weak, release consistency)
- specific to processor architectures (e.g. Alpha, PowerPC, Sparc, IA-64)

relaxed memory consistency models complicate programming and porting because of

- various instruction reordering optimizations
- atomicity constraints of memory operations
- variations in memory semantics

Sparc V9 supports PSO (partial store order), RMO (relaxed memory order) and TSO (total store order) models
➤ Today’s SPARC systems use TSO almost exclusively because switching to conceptually more effective PSO or RMO would require redesigning the operating system.

➤ Due to those difficulties some argue that future systems should implement sequential consistency in hardware because the performance boost does not justify the burden placed on system software programmers.

➤ The authors believe that a compiler can insert fence instructions and thus allow programmer to treat compiler and underlying architecture as a sequentially consistent system and still profit from the performance advantage.
Figure 2: The compiler
In this paper they describe the compiler with these characteristics:

- the compiler presents to its users a sequentially consistent view of the target machine, irrespective of the memory model the machine implements
- the compiler reorders operations when such reordering is in agreement with the sequential consistency model

The algorithm used is based on Shasha and Snir’s [1989] delay set analysis

The algorithm inserts fence instructions by identifying memory barrier nodes and minimizing their number

A notion of dominators with respect to a node is introduced in a control flow graph to identify memory-barrier nodes
Sequential Consistency and Delay Set Analysis

- multithreaded program on a uniprocessor executes in a program order — the order of execution specified by the source program

- sequential consistency is a natural and intuitive extension of the memory behaviour of uniprocessors to multiprocessors

**Definition 1.** 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 in this sequence in the order specified by its program.
Delay set analysis (Shasha and Snir [1989]) finds a minimal set of execution orderings that must be enforced to guarantee sequential consistency.

Let $P$ be a relation that represents the program order of operations. Two operations $m$ and $n$ are in relation $P$ if there is a control flow path from $m$ to $n$ ($mPn$).

Let $C$ be a conflict relation on a variable access. The conflict relation is a the set of all pairs $(v_i, v_j)$ of 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 the conflict relation compared to that presented by Shasha and Snir.

A critical cycle is a cycle in $P \cup C$ with the properties:

1. It contains at most two memory operations from the same thread and they are consecutive in it.
2. It contains either zero, two, or three accesses to each shared variable and they are consecutive in it. The possible configurations are $read x \rightarrow write x$, $write x \rightarrow read x$, $write x \rightarrow write x$, or $read x \rightarrow write x \rightarrow read x$.

A delay relation $D$ between two operations $u$ and $v$ forces $v$ to wait until $u$ completes execution.

$D$ enforces sequential consistency if it contains all $P$ edges in all the critical cycles.
If \( D \) consists of exactly all \( P \) edges in all critical cycles, then \( D \) is minimal.

For example, code shown in Figure 3 gives an outcome \( X=0 \) and \( Y=1 \) for the execution ordering \( b1 \ a2 \ b2 \ a1 \).

The code in Figure 4 can output \( x=1, \ y=1, \ X=0, \) and \( Y=1 \) for the execution ordering \( a1 \ a1 \ a2 \ b2 \ b1 \ b1 \) among others.
\[ \begin{align*}
x & = 0 \\
y & = 0 \\
X & = 0 \\
Y & = 0 \\
\text{cobegin} & \\
\quad a1: & \ x = 1 \\
\quad b1: & \ y = 1 \\
\mid | & \\
\quad a2: & \ Y = y \\
\quad b2: & \ X = x \\
\text{coend} & \\
\text{print} & \ X, \ Y \\
\end{align*} \]

(Figure 3: An example of critical cycles and delays.)
\[
\begin{align*}
x &= 0 \\
y &= 0 \\
X &= 0 \\
Y &= 0 \\
\text{cobegin} \\
&\quad \text{do } i = 1, 2 \\
&\quad \quad a1: X = x \\
&\quad \quad b1: Y = y \\
&\quad \text{end do} \\
\| \\
&\quad a2: x = 1 \\
&\quad b2: y = 1 \\
\text{coend} \\
\text{print } x, y, X, Y
\end{align*}
\]
Relaxed Memory Consistency Models

➤ Reduced restrictions on overlapping and reordering of memory operations.

➤ Memory access operations can be reordered if they are referencing different locations and satisfy the ordering constraints specific to each model.

➤ Causal consistency (Hutto and Ahamad [1990]) requires only that potentially causally related writes must be seen by all processes in the same order. Concurrent writes may be seen in different order on different machines.
PRAM consistency (Lipton and Sandberg [1988]) ensures that all writes done by a single process are seen by all other processes in the order in which they were issued, as if they were in a pipeline. Writes from different processes may be seen in different order by different processes.

Processor consistency (Goodman [1988]) requires that the order of writes issued by one processor be the same as the order specified by the program. However, writes from two different processors may be observed in an order that is not identical.

- The difference between processor and PRAM consistency models is the requirement for memory coherence in the processor consistency model.
- Memory coherence means that for every memory location, \( x \), there must be a global agreement about the order of writes to \( x \). Writes
to different locations need not be viewed in the same order by different processes.

- Weak consistency model (Dubois, Scheurich, and Briggs [1986]) distinguishes between ordinary and synchronization accesses. The memory is required to become consistent on synchronization accesses only.
  - Before any read or write access is allowed, all previous synchronization accesses must be performed.
  - Similarly, before a synchronization access is allowed, all ordinary read and write accesses must be performed.
  - Synchronization accesses are sequentially consistent with respect to one another.
  - We have to explain the exact meaning of the word *performed* in this context.
A read is said to have been performed when no subsequent write can affect the value returned.
A write is said to have been performed when all subsequent reads return the value written by that write.
A synchronization access is said to have been performed when all shared variables had been updated.

The extension of processor and weak consistency models was release consistency (Gharachorloo [1990]) and it was widely used in practical implementations of DSM. The basic idea of the release consistency model lies in a fact that memory synchronization involves two basic operations.

1. All changes that a process has made to the memory are being propagated to other processes.
2. All changes that other processes have made to the memory are being propagated to the process.
A closer observation shows that these two operations need not be performed in the same time. The first one has to be done only when a process exits a critical section. The second one, has to be done only when a process enters a critical section.

Hence, the release consistency model uses two synchronization primitives: acquire and release. Acquire tells the system that the process is about to enter a critical section. Release tells the system that the process is about to exit the critical section.

Release consistency can be also achieved by using barriers. When a process encounters a barrier during its execution it waits until all other processes in its group reach the same barrier. In terms of the release consistency model, the completion of the execution of the last waited process up to barrier is equivalent to release. Departure from a barrier is equivalent to acquire.
Consequently, the release consistency model requires that:

1. All previous *acquires* performed by a process must be performed before an ordinary data access to the memory is allowed.
2. All previous ordinary data accesses to the memory by a process must be performed before a *release* access is allowed.
3. All *acquire* and *release* accesses must be processor consistent.

- It is interesting that sequential consistency is neither required nor needed for synchronization accesses.

Keleher, Cox, and Zwaenepoel [1992] have modified the release consistency model to *lazy release consistency*. The difference between the lazy and eager (standard) release consistency is that in the lazy consistency model nothing is sent anywhere at the time of *release*. Instead, the process performing *acquire* has to get the most recent values of the variables from the node that has the most up-to-date
values. A timestamp protocol can be used to determine which variables have to be transmitted.

➤ Finally, Bershad, Zekauskas, and Sawdon [1993] describe the entry consistency model. It requires that each shared variable be associated with some synchronization variable such as a lock or barrier. Modification of an ordinary synchronization variable is postponed till the next acquire of the synchronization variable that guards it. The traffic is significantly decreased since only the changes for associated variables need to be propagated at the moment of an acquire. The parallelism is also increased, or, looking from a different perspective, the latency is decreased, since there is no waiting for the completion of the unrelated acquires. However, the programming of such a system becomes increasingly complicated.

➤ Lee and Padua are considering only processor, weak, and release
consistency in this paper.

➤ Release consistency divides shared memory operations to several categories shown in Table 1

<table>
<thead>
<tr>
<th>shared</th>
<th>competing unordered conflicting accesses</th>
<th>noncompeting $R_{NC}$, $W_{NC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>synchronization</td>
<td>non-synchronization</td>
</tr>
<tr>
<td>acquire $S_A$</td>
<td>release $S_R$</td>
<td>$R_{NS}$, $W_{NS}$</td>
</tr>
</tbody>
</table>

➤ The ordering constraints of each memory consistency model in a single processor are shown in Table 2
<table>
<thead>
<tr>
<th>Processor consistency</th>
<th>$\mathcal{R}<em>{DR}, \mathcal{R}</em>{DW}, \mathcal{W}<em>{DW}, \mathcal{S}</em>{DR}, \mathcal{S}<em>{DW}, \mathcal{R}</em>{DS}, \mathcal{W}<em>{DS}, \mathcal{S}</em>{DS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak ordering</td>
<td>$\mathcal{S}<em>{DR}, \mathcal{S}</em>{DW}, \mathcal{R}<em>{DS}, \mathcal{W}</em>{DS}, \mathcal{S}_{DS}$</td>
</tr>
<tr>
<td>Release consistency ($\text{RC}_{pc}$)</td>
<td>$\mathcal{S}<em>{\mathcal{A}DR}, \mathcal{S}</em>{\mathcal{A}DW}, \mathcal{R}<em>{\mathcal{DS}</em>{\mathcal{R}}}, \mathcal{W}<em>{\mathcal{DS}</em>{\mathcal{R}}}, \mathcal{R}<em>{\mathcal{NS}}\mathcal{DR}</em>{\mathcal{NS}}, \mathcal{R}<em>{\mathcal{NS}}\mathcal{DW}</em>{\mathcal{NS}}, \mathcal{W}<em>{\mathcal{NS}}\mathcal{DW}</em>{\mathcal{NS}}, \mathcal{S}<em>{\mathcal{A}DS</em>{\mathcal{A}}}, \mathcal{S}<em>{\mathcal{A}DS</em>{\mathcal{R}}}, \mathcal{S}<em>{\mathcal{R}DS</em>{\mathcal{R}}}, \mathcal{R}<em>{\mathcal{NS}}\mathcal{DS}</em>{\mathcal{A}}, \mathcal{S}<em>{\mathcal{R}DW}</em>{\mathcal{NS}}$</td>
</tr>
</tbody>
</table>

$\mathcal{R} = \mathcal{R}_{\mathcal{NS}} \cup \mathcal{R}_{\mathcal{NC}}$ and $\mathcal{W} = \mathcal{W}_{\mathcal{NS}} \cup \mathcal{W}_{\mathcal{NC}}$
Hiding Relaxed Memory Consistency

Preserving Sequential Consistency Using Delays

➤ Suppose that $u \mathcal{D} v$, $v \mathcal{D} w$, and $u \mathcal{D} w$. It is not necessary to enforce $u \mathcal{D} w$ since it is implied by $u \mathcal{D} v$ and $v \mathcal{D} w$.

➤ In general, for a given $\mathcal{D}$, it is sufficient to enforce the transitive reduction of $\mathcal{D}$ (proven by Shasha and Snir).

➤ A transitive reduction of $\mathcal{D}$, $\mathcal{D}^{tr}$, is the minimum relation such that $(\mathcal{D}^{tr})^+ = \mathcal{D}^+$. 

➤ Each memory consistency model has its own constraints and there are delays $\mathcal{D}_o$ that can be enforced by these constraints.
Then, $D_m = ((D \cup D_o)^+)^{tr} - D_o$ is the minimal delay relation we want to find.

Only the delays in $D_m$ need to be implemented with special instructions (fences and synchronization operations).

**Definition 2.** 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.

If the architecture supports weaker fences (e.g. Sparc) even more efficient execution can be attained.
Figure 5: Inserting fence instructions.
Identifying Memory-Barrier Nodes

➤ Starting here, weak ordering is used for the examples.

➤ To enforce a delay $u \mathbf{D_m} v$ we identify as a synchronization instruction a node $y$ which always executes after $u$ and before $v$ whenever $u$ and $v$ execute.

➤ If a node contains a synchronization instruction the authors call it a memory-barrier node.

➤ A condition for finding node $y$ that enforces $u \mathbf{D_m} v$ 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.
Figure 6: The fence $F$ is enough to enforce both of the delays $nD_w$ and $uD_u$. 
To find memory-barrier nodes, the authors introduce the notion of *dominators with respect to a node*.

A node $n$ dominates a node $m$ ($n \text{ dom } m$) if every control flow path from the program entry node to $m$ goes through $n$.

A node $n$ postdominates a node $m$ ($n \text{ pdom } m$) 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 3.** A node $s$ dominates a node $v$ with respect to a node $u$ if there exists some control flow path from $u$ to $v$ and if every control flow path from $u$ to $v$ goes through $s$. This relation is denoted by $s \text{ dom}_u v$ and the set of all nodes that dominates $v$ with respect to $u$ is denoted $\text{dom}_u[v]$.
➤ 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 \( u \).

➤ The authors use an iterative data flow framework to compute dominators with respect to a node \( u \), which is similar to the iterative algorithm for classical dominator, except that the node \( u \) is treated as the program entry node.
Minimizing the Number of Memory-Barrier Nodes

➤ The ultimate objective is to minimize the number of memory-barrier nodes executed by the program.

➤ However, reducing the total number of memory-barrier nodes is a good first approximation to the above.

➤ The authors prove that this minimization is an NP-hard problem.

MIN NODES: An instance \((G, D_m, k)\) of MIN NODES consists of a directed graph \(G\), a finite set \(D_m\) of delays, and a positive integer \(k\). The problem is to find a set \(S\) of size \(k\) whose members enforce all the delays in \(D_m\). A node \(n\) enforces a delay \(uD_m v\) if \(n \in \text{dom}_u[v]\).
Lemma 1. **MIN NODES** belongs to the class NP.

Proof. Suppose that a nondeterministic algorithm guessed S. We need to check in polynomial time whether elements of S enforce all delays in $D_m$ and whether its size is $k$. We know $S \subseteq \bigcup_{(u,v) \in D_m} \text{dom}_u[v]$. Members of S can be counted in linear time to determine whether its size is $k$. To check whether a node $n \in S$ enforces a delay $uD_m v$, first check if there is a path from $u$ to $v$ that goes through $n$. Then we take the node $n$ and its incident edges out of $G$ to form $G'$. If there is no path from $u$ to $v$ in $G'$ then $n \in \text{dom}_u[v]$ and enforces the delay $uD_m v$. This check can be done in polynomial time. □

Lemma 2. **MIN NODES** is NP-hard.

Proof. By a polynomial reduction function that maps every instance of the VERTEX COVER problem to an instance of the MIN NODES problem. An instance of the VERTEX COVER problem consists of an
undirected graph $G_{VC} = (V_{VC}, E_{VC})$ and a positive integer $k$. 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 $S$.

Identify each element $n \in V_{VC}$ by a unique positive integer in $\{1, \ldots, |V_{VC}|\}$. Now, we can define $G$ and $D_m$ in such a way that there is a one-to-one correspondence between an edge in $E_{VC}$ and a delay in $D_m$:

$$G = (N, E), \text{ where}$$

$$N = V_{VC} \cup \{\text{entry, exit}\}$$

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

$$D_m = \{(u, v) | \{u, v\} \in E_{VC} \land u \leq v\}.$$
Obviously, this can be done in polynomial time.

**Figure 7**: An instance of the **VERTEX COVER** problem and the corresponding **MIN NODES** problem.
Using the example in Figure 7 it can be shown that the transformation is indeed a reduction: if a set of \( k \) memory-barriers that enforces \( D_m \) is found, then the corresponding set of vertices will be a solution of size \( k \) to the \textsc{Vertex Cover} problem. On the other side, for a set of \( k \) vertices satisfying \textsc{Vertex Cover} problem, the corresponding set of nodes will enforce all the delays in \( D_m \) and thus solve \textsc{Min Nodes}. \( \square \)

**Theorem 1.** \textsc{Min Nodes} is \textsc{NP}-complete.

**Proof.** From the previous two Lemmas. \( \square \)

- Hence, the approximation algorithm.

- It is a slight modification of the greedy approximation algorithm for the optimization version of the \textsc{Minimum Cover} problem.
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 element in \(C\). The problem is finding the minimum subset \(S \subseteq C\) whose members cover all elements in \(X\) (i.e. \(\bigcup_{c \in S} c = X\)).

**MIN NODES** is converted into **MINIMUM COVER** as follows:

\[
\begin{align*}
X &= D_m \\
C &= \{C_n | n \in \bigcup_{(u,v) \in D_m} \text{dom}_u[v]\},
\end{align*}
\]

where \(C_n\) is the set of all delays that are enforced by a node \(n\).
Let $D$ be the delays found by the delay set analysis and $D_o$ be the delays enforced by the ordering constraints.

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

for each $u \in D_m$ do
  Compute $\text{dom}_u[v]$.
end for

Compute the set $M$ of of nodes using the MIN NODES algorithm.
Label each node in $M$ with Sync.

**Figure 8:** Minimizing the number of memory-barrier nodes and enforcing delays in the weak ordering model.
The time complexity of the algorithm is $O(|D_m|^2|U|)$.

The solution found is not more than $H(\max |S| : S \in C)$ times as large as an optimal solution, where $H(n)$ is the $n$th harmonic number.

It is desirable to mark as a memory barrier a node that is as close to the sink of delay as possible to maximize the opportunity for reordering and overlapping.

It is also desirable that a node be located in the less frequently executed path.
Case 1  Case 1’  Case 2  Case 3  Case 4  Case 5

FIGURE 9: Profitability of inserting a fence instruction for a delay.
FIGURE 10: Limitations of the algorithm based on dominators with respect to a node.
Figure 11: Inserting fence instructions at a node $n$ that is both source and sink of more than one delay.
Related Work

➤ Despite the importance of the topic there is no previous work that deals with the compiler technique.

➤ Adve and Gharachorloo’s programmer-centric specification approach.

➤ Shen’s Commit-Reconcile & Fences (CRF) memory consistency model uses term rewriting and gives clear algebraic semantics. Serves as an interface between programmer-centric models and real implementation.

➤ Java supports concurrent programming with a relaxed memory model.
Krishnamurthy and Yelick developed a back path finding algorithm to find delays in SPMD programs. Time complexity $O(n^3)$, where $n$ is the number of remote memory access operations.

Midkiff proposed a delay set analysis technique to deal with programs with loops and arrays.

Li and Abu-Sufah developed an algorithm to reduce the number of synchronized memory references to shared data in doacross loops.
Conclusions

➤ A compiler technique based on delay set analysis shifts the burden from a programmer.

➤ Sequential consistency insured through memory barrier nodes for each delay found by delay set analysis.

➤ Introduced the concept of dominance with respect to a node.

➤ Compiler can freely reorder in between memory barrier nodes.

➤ The algorithm is not able to find instances when delay need not be enforced for sequential consistency. Enforcing a delay is a sufficient but not necessary condition.