Compiler Techniques For High Performance Sequentially Consistent Java Programs *

Zehra Sura
IBM Watson Research Center
Yorktown Heights, NY 10598
zsura@us.ibm.com

Xing Fang
Purdue University
West Lafayette, IN 47907
xfang@purdue.edu

Chi-Leung Wong
University of Illinois
Urbana, IL 61801
cwong1@uiuc.edu

Samuel P. Midkiff
Purdue University
West Lafayette, IN 47907
smidkiff@purdue.edu

Jaejin Lee
Seoul National University
Seoul, Korea 151-742
jlee@cse.snu.ac.kr

David Padua
University of Illinois
Urbana, IL 61801
padua@uiuc.edu

ABSTRACT

The rise of Java, C#, and other explicitly parallel languages has increased the importance of compiling for different software memory models. This paper describes co-operating escape, thread structure, and delay set analyses that enable high performance for sequentially consistent programs.

We compare the performance of a set of Java programs compiled for sequential consistency (SC) with the performance of the same programs compiled for weak consistency. For SC, we observe a slowdown of 10% on average for an architecture based on the Intel Xeon processor, and 26% on average for an architecture based on the IBM Power3.

Categories and Subject Descriptors: D.3.2 [Programming Languages]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.4 [Programming Languages]: Processors—*Compilers*

General Terms: Experimentation, Languages, Performance

Keywords: Java, memory consistency, synchronization, multithread

1. INTRODUCTION

The effect of memory models on programmer's productivity and program performance is perhaps one of the most difficult and poorly understood issues in shared-memory parallel programming. In this paper, we explore an important part of this problem: what compiler techniques are needed to minimize the performance impact of adopting sequential consistency, a memory model with potential advantages for programmer's productivity.

When the order of shared memory accesses is not completely defined by synchronization operations, the outcome of a parallel program can vary depending on the actual order in which these accesses occur. A memory consistency model defines the order of shared memory accesses that different threads must appear to observe. Thus, memory models are necessary to determine what constitutes a correct execution, and programmers of parallel machines must have a clear understanding of the memory model of their execution environments. The importance of memory models has become more evident in the recent past with the advent of Java, which includes a memory model as part of its specification. This memory model, however, has been the subject of much controversy. Many computer scientists are of the opinion that the Java memory model must be improved. Several proposals have been studied and progress has been made. However, what is perhaps the most natural memory model, sequential consistency, has not been considered as an alternative mainly because of concerns with performance.

Sequential consistency (SC)[17] is often considered to be the simplest and most intuitive memory model[13]. It requires that all shared memory accesses appear to occur in the order in which they appear in the program. However, many languages present the programmer more relaxed memory models, such as weak ordering and release consistency[2], reflecting the memory model of the target machine.

Relaxed models, like that of Java, impose fewer constraints than SC on the order of shared memory accesses. This allows more instruction re-ordering, increasing the potential for instruction level parallelism and better performance. However, for this same reason, it is more difficult to reason about the order of events in programs that follow a relaxed model. Thus, adopting SC should improve the programmer's productivity.

The issues facing programmers with different memory models are illustrated by the *busy-wait* construct that syn-

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-0081265 and CCR-0313033, and the IBM Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the IBM Corporation.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

```
Thread 1                    Thread 2
   ⋮                           ⋮
Data = ...;             while (Flag==0) wait;
Flag = 1;                   ... = Data;
```

Figure 1: Busy-wait Synchronization Example

chronizes accesses to `Data` in Figure 1. The code fragment in Figure 1 provides busy-wait synchronization in a sequentially consistent system. However, for most relaxed memory consistency systems, this busy-wait synchronization will not work as expected. For a language that supports a relaxed memory model, the compiler can move `Flag = 1` prior to "`Data = ...`", breaking the desired synchronization. The target architecture may also break the synchronization if it implements a relaxed model that re-orders the two write instructions.

The example also illustrates challenges faced by a memory model aware compiler. To generate efficient and correct code, it must determine which accesses in each thread are to memory locations also accessed in another thread, and which of those accesses may not be re-ordered. It must determine when common optimizations like dead code elimination, common subexpression elimination, and register allocation must be restricted[24]. For example, allocating `Flag` to a register will cause the `while` loop to never terminate when the value of `Flag` assigned to the register is zero.

Although, as mentioned above, SC seems to be the ideal choice for programmer productivity[13], it is not clear how the re-ordering restrictions of SC impact performance. Determining the magnitude of this impact is not an easy task because SC does not explicitly prohibit re-ordering of shared memory accesses in a thread; it only requires that an execution maintain the illusion that no shared memory accesses in a thread have been re-ordered. Thus, a compiler could analyze the program to identify re-orderings that do not break the illusion of SC. As a result, performance depends not only on the re-ordering restrictions of SC relative to those of relaxed models, but also on how accurately the compiler identifies the orders that must be enforced.

Well-synchronized programs include explicit synchronization such that the set of valid shared memory access orders is the same for SC and relaxed models. In practice, most programs are written to be well-synchronized. This means that a compiler with perfect analysis capabilities should be able to produce code for a program that both follows the SC model and is close to the performance of code generated for a relaxed model. However, analyses are not perfect. So, performance depends on the precision with which a compiler can determine the shared memory access orders that must be enforced to honor SC.

The objective of the study reported in this paper was to determine if the performance of SC programs can approximate that of programs which follow a relaxed memory model, and to determine the compiler techniques needed to achieve this. This paper describes the compiler analysis techniques that we developed for this study and presents performance results which indicate that in many cases SC programs can achieve performance that is very close to the performance of relaxed consistency programs.

The outline of the rest of this paper is as follows. In Section 2, we present the overall design of our compiler. We describe our analysis algorithms in Sections 3 and 4. We briefly discuss the complexity of these algorithms in Section 5. In Section 6, we give experimental data comparing the performance of SC with weak consistency. We discuss related work in Section 7. Finally, in Section 8, we present our conclusions.

2. OUR COMPILER SYSTEM

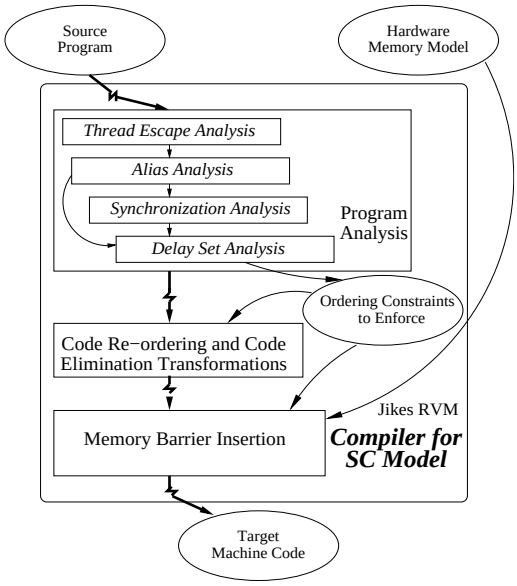


Figure 2: Components of Our Memory-Model Aware Compiler

Figure 2 shows the components of our compiler system, which is an extension of the Jikes Research Virtual Machine (Jikes RVM) from IBM[3]. The Jikes RVM is a Java virtual machine that uses dynamic just-in-time compilation to translate Java classes into executables. It includes a base compiler as well as an optimizing compiler that can be directed to perform various levels of optimizations. The components in the figure are the analyses and transformations that we add to the Jikes RVM to support SC.

Given a source program, the *program analysis* component determines shared memory access orderings that must be enforced so that SC is not violated. The *code re-ordering and code elimination transformations* component restricts optimizations so that when compiling for SC an access in a pair whose order needs to be enforced is not re-ordered or eliminated. Jikes RVM compiler optimizations that may reorder or eliminate shared memory accesses include loop invariant code motion, loop unrolling, common subexpression elimination, and redundant load elimination. The *memory barrier insertion* component also uses the analysis results, along with knowledge of the memory model of the target architecture, to generate code that enforces the required access orders at the hardware level. It does this by emitting machine instructions (called *sync* on the IBM Power3 architecture, and *fence* on the Intel Xeon architecture) to enforce access orders not enforced by the hardware but required by SC[20]. In our system, we use the local-optimized version of

memory barrier insertion described in [9]. The rest of this section describes the program analysis components in our compiler.

Delay set analysis determines orderings within a thread that must be enforced to honor the semantics of SC. We describe our delay set analysis algorithm in Section 3. To perform delay set analysis, we require several other supporting analyses: synchronization analysis, thread escape analysis, and alias analysis.

Synchronization analysis determines orderings that are enforced by explicit synchronization in the program. This helps improve the precision of delay set analysis. We describe our synchronization analysis in Section 4.

Thread escape analysis identifies shared memory accesses that reference an object that may be accessed in more than one thread. Only operations with thread escaping memory accesses need to be considered when performing delay set analysis. Our implementation uses a simple type-based thread escape analysis: if a reference to an escaping object is written into a memory location of type T, then all accesses to any memory location of type T may be considered to be escaping. The analysis is iterative, partially context-sensitive, and flow-insensitive[30].

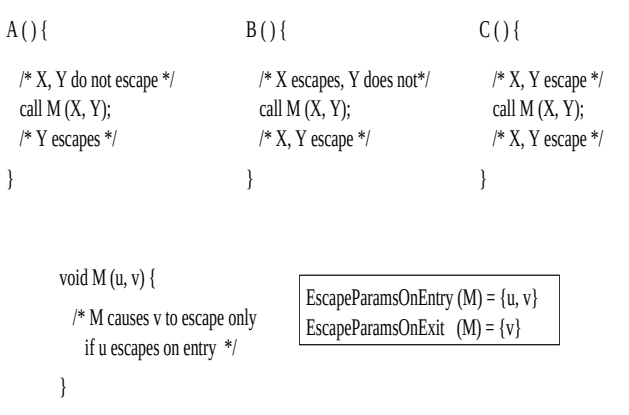


Figure 3: Different Calling Contexts of Methods for Escape Analysis

Figure 3 illustrates how our analysis is partially context-sensitive. Method M takes two parameters u and v, and the actions of M are such that it causes v to escape only when the reference passed in u escapes. Method M is called from three points in the program:

1. Method A calls M with neither parameter initially escaping.
2. Method B calls M with u initially escaping but v not escaping.
3. Method C calls M with both parameters initially escaping.

Our analysis computes $EscapeParamsOnEntry(M)$ to be the set of parameters that may be escaping when M is invoked in any context in the program. It also computes $EscapeParamsOnExit(M)$ to be the set of parameters that may escape as a result of some execution of M. We correctly determine that Y escapes after the call to M in B returns, and that X does not escape after the call to M in A returns. However, we conservatively determine that Y escapes after the call to M in A returns.

Alias analysis determines if two accesses may refer to the same memory location. Delay set analysis uses this information to determine conflicting pairs of accesses that allow threads to communicate by accessing the same memory location. Alias analysis is needed only for thread-escaping accesses. We take advantage of strong typing in the Java language, and use a type-based alias analysis. In our implementation, any two accesses that refer to a value of the same type may access the same memory location. We do not use array dependence analysis, so all accesses to elements of an array are considered to access every element in the array.

3. DELAY SET ANALYSIS

Delay set analysis computes a *delay set*, i.e. a set of ordered pairs of memory accesses (x, y) such that y must be delayed until x has completed.

3.1 Background

In [29], Shasha and Snir show how to find the minimal delay set. Their analysis uses a graph that has a node for each shared memory access and two types of edges: *program edges* and *conflict edges*.

Program edges are directed and represent the linear ordering that is assumed in SC between shared memory accesses in the same thread.

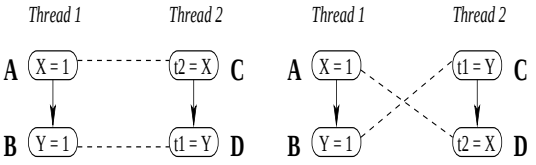
Conflict edges connect nodes that may access the same memory location such that at least one of the accesses is a write. In general, conflict edges are undirected, and the outcome of a program execution may differ depending on which of the accesses connected by a conflict edge executes first.

Consider a program edge from an access A to an access B. Suppose there are no intra-thread dependences between A and B and that B executes first. If no other instruction accesses memory location A or B, then this re-ordering has no effect on the outcome of the program. For the re-ordering of A and B to have an effect, there must be accesses in other threads that conflict with A and B in such a way that the re-ordering is recorded by the state of the program and later reflected in its output. In terms of the graph considered for delay set analysis, [29] shows that this translates to a "critical" path from B to A in the graph, such that the path contains:

- at most two accesses from any thread, and these accesses are adjacent in the path.
- either zero, two, or three accesses to any given memory location, and these accesses are adjacent in the path.

Therefore, if the graph contains a program edge from A to B, as well as a critical path from B to A (i.e. a critical cycle), then re-ordering the two accesses A and B may affect the outcome of the program.

For the example graphs in Figure 4, solid edges are program edges while dashed edges are conflict edges. X and Y refer to shared memory locations that are initially zero. Assuming SC, the outcome $t1 = 1$ and $t2 = 0$ is valid for the example in Figure 4(a), but not for the example in Figure 4(b). There is a critical cycle (A,B,C,D,A) in the graph in Figure 4(b) that causes the program edge from A to B to be a delay edge. Thus, A and B may not be re-ordered.



(a) A and B may be re-ordered

(b) A and B may not be re-ordered

Figure 4: Re-orderings Allowed Under SC

The graph in Figure 4(a) contains no cycle, and A and B may be re-ordered in this case.

Shasha and Snir's algorithm[29] finds all critical cycles in the graph. Each program edge in a critical cycle is a *delay edge*, and it represents an intra-thread order that must be enforced for any execution of a program to honor SC semantics. In our analysis, we use an approximate test to determine if a program edge is a delay edge. For a program edge from A to B, if no path exists from B to A that begins and ends with a conflict edge, where the conflicts are between accesses in different threads, then the program edge is not a delay edge.

3.2 Our Simplified Delay Set Analysis

Precise delay set analysis finds the minimal set of delay edges that need to be enforced in a program. However, it cannot be performed in a compiler for a general-purpose language because of control flow, imperfect memory disambiguation, or an unknown number of program threads.

Our goal is to perform delay set analysis that is approximate and fast, but good enough to generate code that performs well for SC. In our simplified analysis, we consider program edges one at a time, and apply a simple, conservative test to determine if it *cannot* be a delay edge. This test checks to see if the end-points of a potential cycle exist. If they do, we conservatively assume that the complete cycle exists, without verifying if it actually does exist in the graph.

To test if a program edge from node A to node B may be a delay edge (see Figure 5), we determine if there exist two other nodes X and Y such that:

1. a conflict edge exists between A and X, and
2. a conflict edge exists between B and Y, and
3. A and X may occur in different threads, and
4. B and Y may occur in different threads, and
5. A may occur after X and Y, and
6. B may occur before X and Y, and
7. Y may occur before X.

If X and Y exist, then we conservatively assume that a path from B to A exists that completes a cycle in the graph, and so the program edge from A to B is a delay edge. If no such nodes X and Y exist, then it follows from the results in [29] that the program edge from A to B cannot be a delay edge.

Note that since our analysis is conservative we independently test each of the orderings enumerated above.

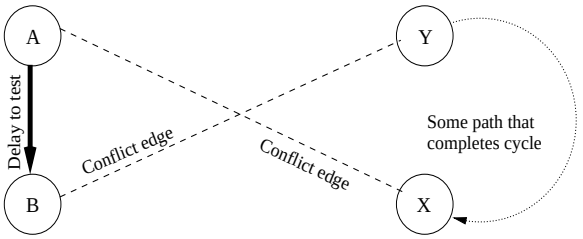


Figure 5: Simplified Cycle Detection

Without synchronization analysis, the check for a delay edge cannot accurately test for the last three conditions enumerated. We use synchronization analysis, described in Section 4, to refine delay set analysis, and test all the conditions enumerated.

3.3 Implementation

Thread escape analysis identifies shared memory accesses that refer to thread escaping locations. These accesses are the nodes in the graph considered for delay set analysis. Nodes may represent multiple accesses in the program execution. In our implementation, there is a node in the graph for each method call, and this node represents all accesses that may be performed when the call invokes a method. When nodes represent multiple accesses, a program edge or conflict edge exists between two nodes A and B if it exists between $\hat{A}$ and $\hat{B}$, where $\hat{A}$ is any access represented by A, and $\hat{B}$ is any access represented by B.

For a node N, the set $ConflictMethods(N)$ is the set of methods that contain an access potentially conflicting with N. We use type-based alias analysis to determine conflicting accesses. Java is a strongly typed language, and each memory access refers to a value of a specific type. However, due to polymorphism and dynamic class loading, a static instance of a memory access may correspond to multiple dynamic accesses that refer to values of different types. Using interface and class hierarchy information, we can identify for each access, a set of types T such that the access may only refer to a value of type T.

DEFINITION 3.1. *For a method* M*, $DirectWr(M)$ is the set of all types T such that some access in* M *may write a value of type T.*

DEFINITION 3.2. *For a method* M*, $AllWr(M)$ is the set of all types T such that some access in* M*, or in another method invoked indirectly when* M *executes, may write a value of type T.*

The sets $DirectRd(M)$ and $AllRd(M)$ are analogous.

DEFINITION 3.3. ConflictMethods(N)

$= \{M \mid T \in DirectWr(M)\}$;
if N is a read access that may refer type T.

$= \{M \mid T \in DirectWr(M)$ *or* $T \in DirectRd(M)\}$;
if N is a write access that may refer type T.

$= \{M \mid \exists\ T, (T \in AllRd(X)$ *and* $T \in DirectWr(M))$ *or*
$(T \in AllWr(X)$ *and*
$(T \in DirectWr(M)$ *or* $T \in DirectRd(M)))\}$;
if N is a method call that invokes method X.

We individually test each program edge to check if it should be marked as a delay edge. To test for a delay edge

from node A to node B, we compute $ConflictMethods(A)$ and $ConflictMethods(B)$. If either set is empty, then the edge from A to B is not a delay edge. Otherwise we assume that there is a delay edge from A to B, unless synchronization information shows that this edge does not exist. Thus, our implementation does not consider each conflict edge separately, but instead summarizes them over all accesses in a given method.

Dynamic Class Loading

For our analysis, we maintain an explicit call graph, and a call site may invoke one of several possible methods, depending on the type of the object on which the call is invoked. Note that we do not handle methods implemented using native code. We perform our inter-procedural analysis over code for all methods possibly executed by the application and available at the point of the analysis. For methods whose code is unavailable, we optimistically assume that the unknown method does not spawn any new threads, and that it contains no shared memory accesses that conflict with an access in another method. Later, when code for the method becomes available due to dynamic class loading, we verify these properties to determine if the analysis results are valid. If an assumption is found to be wrong, we perform the analysis again, invalidate previously compiled methods that may now need extra memory barriers inserted, and recompile these methods to generate correct code. Instead of repeating the entire analysis from scratch, we design it to build upon previous results. In [30], we give details of how we update the call graph, the types accessed by a method, and the synchronization analysis results, and how we determine the set of methods to invalidate.

We use the Jikes RVM method invalidation mechanism to invalidate the code generated for these methods. This ensures that any subsequent calls to the method will use newly generated code. However, this mechanism does not handle the case when a method call has already been invoked by a thread and is currently active on its execution stack. So, our implementation cannot invalidate methods that are currently executing. This is not an issue for the experiments we perform, as no methods on the stack need to be invalidated during execution of our benchmark programs.

We modify the runtime system of the Jikes RVM to detect the first time the application spawns a new thread. Until a thread is spawned, we assume the application is single-threaded, and delay set analysis trivially finds no delay edges. When the runtime system is first invoked by the application to spawn a new thread, it passes control to the optimizing compiler. At that point, our inter-procedural analysis is performed for the first time, and the check for delay edges is no longer trivial.

4. SYNCHRONIZATION ANALYSIS

Synchronization information helps reduce the number of conflict edges in the graph considered for delay set analysis, and thus improves the precision of delay set analysis[16].

In our analysis, we consider the following Java synchronization primitives:

- `synchronized` blocks, used for lock-based synchronization.
- thread `start()` and `join()` calls, used to determine the program thread structure.

Our lock-based synchronization analysis is described in [30]. It helps improve the accuracy of our approximate delay set analysis. In essence, when we search for initial conflict edges that form the end-points of a potential cycle, we can ignore conflict edges that occur between two accesses within synchronized blocks of the same lock object.

In the rest of this section, we describe our thread structure analysis.

4.1 Thread Structure Analysis

Java language semantics guarantee that when a thread is spawned via a thread `start()`, all memory accesses of the creator thread complete before the point where the new thread starts. Also, if a thread T invokes a `join()` call to wait for another thread to terminate, then all memory accesses performed by the terminating thread complete before T continues execution after the `join()`. Our SC compiler enforces these rules when it generates machine code by inserting a fence or a sync instruction before each `start()` call, and after each `join()` call.

We use dataflow analysis to determine orderings due to the thread structure of the program. For each pair (S, P), where S is a thread `start()` call, and P is a point in the program, we determine the relative order of execution of P and all threads started at S. Before we perform our dataflow analysis, we identify all thread `start()` and `join()` calls in the program, and match, whenever possible, the `join()` calls to corresponding `start()` calls.

We say that a `join()` matches a `start()` if the `start()` and `join()` are both invoked on the same object. Our analysis conservatively assumes that two references are to the same object only if they are from the same program variable, this variable can only be accessed by a single thread, and flow analysis can ensure that the variable is not reassigned on any path from the first reference to the second reference. We use thread escape analysis and a conservative intra-procedural flow analysis, and attempt to match a `join()` with a `start()` only if both the `start()` and the `join()` occur in the same method.

If a `start()` is in a loop, then it must be invoked on a distinct object for each iteration of the loop. So, a set of threads is associated with the `start()`. We say that a `join()` matches this `start()` if it is invoked on exactly the same set of objects as the `start()` is invoked on.

When doing the analysis, we consider only those `join()` calls that are matched with some `start()`. It is safe to leave some `join()` unmatched because we then consider fewer synchronizations, which leads to a conservative, but correct, analysis.

We use an inter-procedural inter-thread analysis to compute, for each program point P, the sets $StartsAfter(P)$, $JoinsBefore(P)$, and $Concurrent(P)$. These are *may* sets, not *must* sets. It is conservative, but correct, to include any `start()` in any of these sets.

- For a thread `start()` S and any program point P, $S \in StartsAfter(P)$ if S does not dominate P.
- For a thread `start()` S and any program point P, $S \in JoinsBefore(P)$ if a thread `join()` that matches with S executes on some path from the program entry point to P.
- For a thread `start()` S and any program point P, $S \in Concurrent(P)$ if S executes on some path from the

program entry point to P and a `join()` that matches with S does not execute on that same path after S and before P.

Initially, at the entry point of the application, E, there is a single application thread and no program statements have executed. So, $StartsAfter(E)$ is the set of all start calls in the program, and $JoinsBefore(E)$ and $Concurrent(E)$ are empty sets.

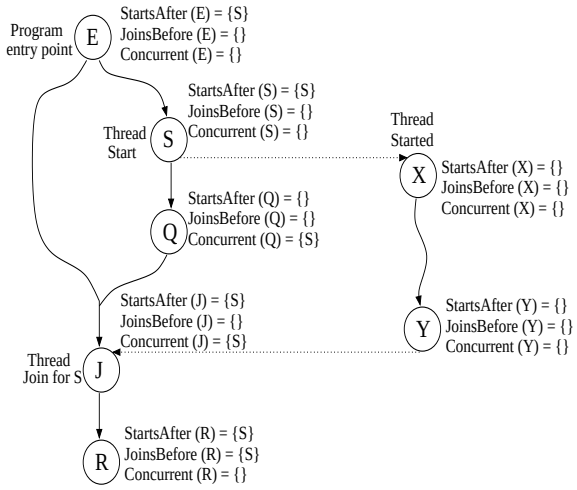


Figure 6: Effect of Thread Start and Join Calls

Consider the program control flow graph in Figure 6. Point E is the program entry point, point S is a thread `start()` call, and point J is a `join()` call that is matched with S. For point Q that occurs after the thread `start()` S, S should be removed from the set $StartsAfter(Q)$, and should be added to the set $Concurrent(Q)$. For point R that occurs after the thread `join()` J, S should be added to the set $JoinsBefore(R)$, and should be removed from the set $Concurrent(R)$.

Point X in Figure 6 is the initial point in the new thread spawned by the `start()` at S. It illustrates the case when S is not contained in any of the sets $StartsAfter(X)$, $JoinsBefore(X)$, and $Concurrent(X)$.

We use dataflow analysis to propagate sets from the application entry point to all reachable points in the program. Figure 7 gives the dataflow equations used. If a point has several predecessors, we propagate the union of the sets of all predecessors. We also propagate information from each `start()` call to the entry point of the `run()` method for the thread type(s) spawned by the `start()`. We iterate over the available code, propagating information until no sets change their values. The analysis must converge because there are a fixed number of `start()` calls in the code, and once a `start()` is added to a set due to propagation, it is never removed from it.

The predecessor relation used in the analysis depends on the memory model. For SC, it is defined by the control flow graph of each method, with the predecessors of the entry point of a method being all call sites for that method.

4.2 Implementation

We efficiently implement dataflow analysis using bitsets and a traversal in topsort order of the strongly connected

$$StartsInMethod(M) = \{s | s \text{ is a } \texttt{start()} \text{ that may execute when } M \text{ is invoked.}\}$$

$$JoinsInMethod(M) = \{s | s \text{ is a } \texttt{start()} \text{ matched to } \texttt{join()}\ j \text{ and } j \text{ must execute when } M \text{ is invoked.}\}$$

$$Kill\_StartsAfter\ (x) = \begin{cases} \{x\} & , x \text{ is a } \texttt{start()} \\ StartsInMethod(M) & , x \text{ is a call to method } M \\ \phi & , \text{otherwise.} \end{cases}$$

$$Gen\_JoinsBefore\ (x) = \begin{cases} \{y\} & , x \text{ is a } \texttt{join()} \text{ that matches } \texttt{start()}\ y \\ JoinsInMethod(M) & , x \text{ is a call to method } M \\ \phi & , \text{otherwise.} \end{cases}$$

$$Gen\_Concurrent\ (x) = \begin{cases} \{x\} & , x \text{ is a } \texttt{start()} \\ StartsInMethod(M) & , x \text{ is a call to method } M \\ \phi & , \text{otherwise.} \end{cases}$$

$$Kill\_Concurrent\ (x) = \begin{cases} \{y\} & , x \text{ is a } \texttt{join()} \text{ that matches } \texttt{start()}\ y \\ JoinsInMethod(M) & , x \text{ is a call to method } M \\ \phi & , \text{otherwise.} \end{cases}$$

$$StartsAfter(S) \cup= \bigcup_{p \in Pred(S)} (StartsAfter(p) - Kill\_StartsAfter\ (p))$$

$$JoinsBefore(S) \cup= \bigcup_{p \in Pred(S)} (JoinsBefore(p) \cup Gen\_JoinsBefore\ (p))$$

$$Concurrent(S) \cup= \bigcup_{p \in Pred(S)} ((Concurrent(p) - Kill\_Concurrent\ (p)) \cup Gen\_Concurrent\ (p))$$

Figure 7: Dataflow Equations

components of each method control flow graph. After the analysis, we preserve only the sets corresponding to method entry points. The sets for each program point are computed intra-procedurally on demand.

Even though our synchronization analysis is inter-procedural and flow-sensitive, it is efficient in practice (Section 6.4). This is because the time it takes for the analysis to converge depends on the number of synchronization constructs in the program. Typically, the number of points in the code where threads are spawned is small compared to the total number of program points. Also, we need not perform a flow-sensitive analysis of methods that do not start or join any threads, either directly or indirectly through another method invoked when the method executes.

We use the results of synchronization analysis to perform delay set analysis. When searching for conflicts, we consider accesses in the same method as an aggregate. The *StartsAfter*(P), *JoinsBefore*(P), and *Concurrent*(P) sets for such an aggregate are the union of the corresponding sets for each access included in the aggregate.

4.3 Using Synchronization Information in Delay Set Analysis

4.3.1 Eliminating Conflict Edges

We say that a thread type satisfies the *single thread constraint* if at most one thread of that type can execute at any given time. In our implementation, T satisfies the single thread constraint if for each point R in the code of T, *Concurrent*(R) does not contain a `start()` S, such that execution of S may directly or indirectly spawn a thread of type T. For efficiency, our analysis aggregates, for each method, synchronization information over all possible accesses that may execute when the method is invoked.

We can ignore conflict edges between shared memory accesses that execute as part of a single thread type, and where that thread type satisfies the single thread constraint. This is because the single thread constraint implies that the two accesses are performed by the same thread, and in our delay set analysis, we search for conflict edges that occur between accesses performed in different threads.

4.3.2 Ordering Program Accesses

Consider two accesses X and Y. We cannot determine whether these two accesses are guaranteed to execute in a fixed order if:

- one of the accesses (say X) may be concurrent with a thread of type T, and
- the other access (say Y) is part of the code that may execute when a thread of type T is spawned, i.e. thread T executes Y, or another thread T' executes Y and T' is spawned as a result of the execution of T.

Otherwise, we may be able to order the two accesses if all instances of X or Y execute before some `start()`, or all of them execute after a `join()` that matches some `start()`.

If an instance of X in thread $T1$ and an instance of Y in another thread $T2$ are ordered by the thread structure of the program, then this order must be evident either from the ordering information for X and Y with respect to the `start()` that spawns $T1$, or from the ordering information for X and Y with respect to the `start()` that spawns $T2$. To determine if there is an order between all instances of X and all instances of Y, we need only consider ordering information for X and Y with respect to `start()` calls S such that S may spawn a thread of type T, and this thread may execute X or Y. We can order $X \rightarrow Y$ (or $Y \rightarrow X$) if the same order is determined with respect to *each* of these `start()` calls S.

In Figure 8, we list all the possible orderings that may be inferred for two program accesses X and Y using their ordering information with respect to one particular `start()`, S.

5. COMPLEXITY

Our synchronization analysis is a forward dataflow analysis with the same complexity as standard dataflow analysis algorithms[12]. It computes sets with at most s elements, where s is the number of thread `start()` calls and synchronized blocks in the program. These sets are represented as bitsets, and it takes $O(1)$ time to test if a `start()` guarantees an order for two nodes, or if a synchronized block allows a conflict edge to be ignored.

The worst case time complexity of our simplified delay set analysis is $O(n * m^2 * s)$, where n is the number of nodes in the graph considered for delay set analysis, and m is the number of methods in the program. We test each intra-procedural program edge in the graph to check if it is a delay edge. Since Java is a structured language, and the maximum outdegree of each node due to control flow is a fixed constant, the number of edges we need to test is $O(n)$.

It takes $O(n)$ time to compute sets *DirectWr*(M), *AllWr*(M), *DirectRd*(M), and *AllRd*(M) for all methods M. Using these sets, it takes $O(m)$ time to compute set *ConflictMethods*(N), for any node N. A delay test between two nodes A and B computes sets *ConflictMethods*(A) and *ConflictMethods*(B). There are m^2 possible choices for a pair (X,Y), $X \in$ *ConflictMethods*(A) and $Y \in$ *ConflictMethods*(B), that may need to be tested for ordering based on synchronization information (i.e. the orderings in (5), (6) and (7) of the checks enumerated in Section 3.2).

Thus, the complexity is $O(n * m^2 * s)$: $O(n)$ delay tests, $O(m^2)$ orderings checked for each delay test, and $O(s)$ time to check an ordering between two nodes based on synchronization information.

6. EXPERIMENTAL RESULTS

We measure the performance penalty incurred when SC, instead of the weak consistency model implemented in the Jikes RVM, is used as the memory model. Weak consistency allows shared memory accesses to be freely re-ordered except when the re-ordering violates single-threaded data dependences or the re-ordering is across an explicit synchronization point in the code.

6.1 Benchmark Programs

Table 1 lists the benchmark programs used in the experiments. The benchmarks include standard multithreaded Java benchmarks (from SPECjvm 98, SPECjbb 2000, and the Java Grande Forum benchmarks) as well as several codes taken from literature, including concurrent implementations of two data structures, hashmaps and queues. These concurrent data structures are expected to be widely used and have been incorporated in the Java standard libraries.

A (Y) => StartsAfter (Y) B (Y) => JoinsBefore (Y) C (Y) => Concurrent (Y)

Access X \ Access Y	A(Y) = {} B(Y) = {} C(Y) = {}	A(Y) = {} B(Y) = {} C(Y) = {S}	A(Y) = {} B(Y) = {S} C(Y) = {}	A(Y) = {} B(Y) = {S} C(Y) = {S}	A(Y) = {S} B(Y) = {} C(Y) = {}	A(Y) = {S} B(Y) = {} C(Y) = {S}	A(Y) = {S} B(Y) = {S} C(Y) = {}	A(Y) = {S} B(Y) = {S} C(Y) = {S}
A(X)={} B(X)={} C(X)={}			**X -> Y**		**Y -> X**			
A(X)={} B(X)={} C(X)={S}			**X -> Y**		**Y -> X**			
A(X)={} B(X)={S} C(X)={}	**Y -> X**	**Y -> X**			**Y -> X**	**Y -> X**		
A(X)={} B(X)={S} C(Y)={S}					**Y -> X**			
A(X)={S} B(X)={} C(X)={}	**X -> Y**	**X -> Y**	**X -> Y**	**X -> Y**				
A(X)={S} B(X)={} C(X)={S}			**X -> Y**					
A(X)={S} B(X)={S} C(X)={}								
A(X)={S} B(X)={S} C(X)={S}								

Figure 8: Orders Inferred From Thread Structure Analysis

Benchmark	Description	Source	# Bytecodes
Hashmap	Microbenchmark for concurrent hashmaps	Uses Doug Lea's ConcurrentHashMap class[18]	24,989
GeneticAlgo	Parallel genetic algorithm	Adapted from the sequential version in [11]	30,147
BoundedBuf	Producer-consumer application	Uses Doug Lea's BlockingQueue class[18]	12,050
Sieve	Sieve of Erastothenes	From an example in [11]	10,811
DiskSched	Disk scheduler using an elevator algorithm	From an example in [19]	21,186
Raytrace	Ray tracing application	Java Grande Forum Multithreaded Benchmarks[1]	33,198
Montecarlo	Monte Carlo simulation	Java Grande Forum Multithreaded Benchmarks[1]	63,452
MolDyn	Molecular dynamics application	Java Grande Forum Multithreaded Benchmarks[1]	26,913
SPECmtrt	Ray tracing application	From the SPECJVM 98 benchmark suite[6]	290,260
SPECjbb00	Middle-layer database server application	SPECJBB 2000[7]	521,021

Table 1: Benchmark Characteristics

6.2 Performance Impact

We measured execution times for the following configurations:

1. *Base_Relaxed*: The default Jikes RVM execution without any of our changes. It uses weak consistency, and freely performs re-ordering optimizations.

2. *Escape_SC*: An implementation of SC that uses only the results of our escape analysis to determine the memory barriers that need to be inserted. It assumes there is a delay edge between any pair of shared memory accesses where both accesses may refer to escaping objects.

3. *Delay_SC*: An implementation of SC that uses all our program analyses to determine the sync and fence instructions that need to be inserted.

We ran our experiments using a FastAdaptiveSemiSpace configuration for the Jikes RVM version 2.3.1, with the level 0 optimizing compiler as the initial compiler. In this configuration, methods that execute for long periods of time are recompiled at higher levels of optimization. The Jikes RVM includes re-ordering optimizations such as loop invariant code motion, loop unrolling, common subexpression elimination, and redundant load elimination.

For each benchmark program listed in Section 6.1, we obtain performance data by averaging over five executions of the program. We obtain performance numbers for two architectures:

- Intel Xeon: A Dell PowerEdge 6600 SMP with 4 Intel hyperthreaded 1.5GHz Xeon processors (each having 1MB cache), and 6GB of system memory, running Linux.

- IBM Power3: An IBM SP 9076-550 with 8 375MHz processors, and 8GB of system memory, running AIX.

6.2.1 Intel Xeon Platform

Figure 9 shows the execution times on the Intel Xeon platform for each of the three configurations, normalized to the *Base_Relaxed* configuration. We observe that our implementation of SC shows a slowdown of 10% on average over weak consistency. This loss in performance is due to extra memory barrier instructions inserted. The performance loss due to inhibiting optimizations is negligible. Delay set analysis and synchronization analysis have a large impact on performance, since the average slowdown is 26.5 *times* when these two analyses are not used.

For 7 out of 10 benchmarks, the slowdowns were 6% or less. The three benchmarks that did not perform as well are:

- *DiskSched*: This application uses busy-wait synchronization implemented with the help of a volatile variable. Our analysis only takes into account locking and thread `start()/join()` calls, and it does not capture this busy-wait synchronization. Therefore, superfluous fence instructions are inserted, and we see a loss in performance.

- *MolDyn*: This application uses several shared arrays that are accessed by all threads. However, each thread accesses only one element of each array. Because our compiler does not perform array dependence analysis, it does not capture this situation and conservatively inserts fence instructions that slow down the application in the case of SC.

- *SPECmtrt*: All threads in this application read data from the same input file. As a result, our conservative escape analysis determines several fields in the Java standard I/O library classes to be escaping. Delay set analysis detects delay edges between accesses to these fields, and fence instructions are inserted to enforce them. This contributes to some of the performance slowdown observed[1].

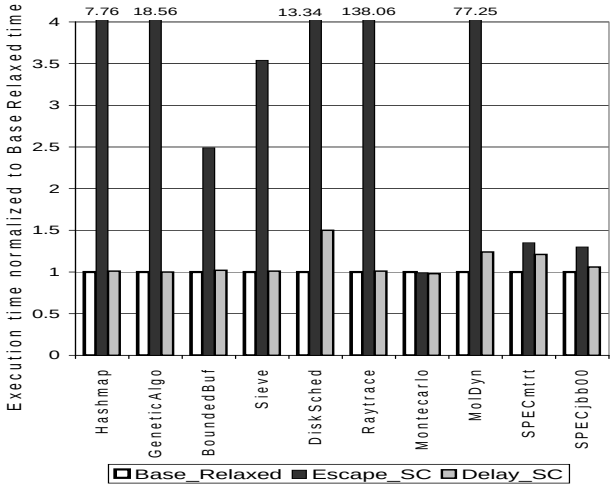


Figure 9: Slowdowns for SC on Intel Xeon

6.2.2 Power3 Platform

Figure 10 shows the execution times for the Power3 platform for each of the three configurations, normalized to the *Base_Relaxed* configuration. We observe that our implementation of SC shows a slowdown of 26% on average over weak consistency. Again, delay set analysis and synchronization analysis have a significant impact on performance, and the average slowdown is 8.21 times when these two analyses are not used.

For 8 out of 10 benchmarks, the slowdowns were 7% or less. The 2 benchmarks that did not perform as well are *DiskSched* and *MolDyn*, that did not perform well on the Intel Xeon platform either.

6.3 Effect of Delay Set Analysis

For each benchmark program, Table 2 shows statistics for the performance of delay set analysis on the Intel Xeon platform, and Table 3 shows statistics for the Power3 platform. The first column in these tables shows the number of times that the test for a delay edge is invoked during compilation. The second column shows the number of times this test identifies a delay edge. Note that we consider only those delay tests that are non-trivial, i.e. tests performed after the application has spawned a thread. The last column shows the percentage of delay tests for which the analysis is able to

[1]As described in [30], more than 50% of the slowdown for *SPECmtrt* is due to adaptive re-compilation for optimization that overlaps with the application execution.

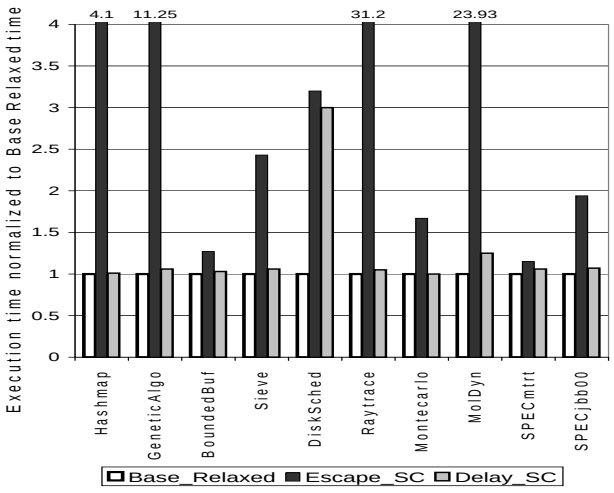


Figure 10: Slowdowns for SC on IBM Power3

Benchmark	Number of Delay Edges		
	Tested For	Need Enforcing	% Removed
Hashmap	16,089	60	99.6%
GeneticAlgo	9,857	367	96.3%
BoundedBuf	8,139	884	89.1%
Sieve	151,699	1,866	98.8%
DiskSched	4,041	58	98.6%
Raytrace	17,418	46	99.7%
Montecarlo	7,156	153	97.9%
MolDyn	13,370	6,924	48.2%
SPECmtrt	174,263	28,434	83.7%
SPECjbb00	1,415,865	115,580	91.8%

Table 2: Delay Edge Counts for Xeon

Benchmark	Number of Delay Edges		
	Tested For	Need Enforcing	% Removed
Hashmap	17,265	347	98.0%
GeneticAlgo	17,664	750	95.8%
BoundedBuf	3,890	824	78.8%
Sieve	5,256	1,121	78.7%
DiskSched	3,296	202	93.9%
Raytrace	78,786	47	99.9%
Montecarlo	16,568	314	98.1%
MolDyn	13,938	6,583	52.8%
SPECmtrt	150,501	19,861	86.8%
SPECjbb00	1,162,591	88,309	92.4%

Table 3: Delay Edge Counts for Power3

determine the absence of a delay edge. On the Intel Xeon platform, our delay set analysis can eliminate on average 90% of the delay edges that the system would otherwise enforce, and on the Power3 platform we eliminate 87.5% of delay edges.

6.4 Analysis Times

The performance numbers in Figure 9 do not include the compile time for an application, except for re-compilation due to adaptive optimization. On the Intel Xeon platform, the compile time is significantly large for three benchmarks, *MolDyn*, *SPECmtrt*, and *SPECjbb*00. *SPECjbb*00 is a server application that, in practice, runs for long periods of time. We expect the compilation overhead to be amortized over this long running time. A similar argument applies to *MolDyn* and *SPECmtrt* if they are used to process large data sets, and execute for a long time.

Table 4 shows the total analysis times (including all *re*-compilations), in seconds, for each benchmark executing on the Intel Xeon platform. Table 5 shows these times for the Power3 platform. The columns in these tables are:

- *Delay*: Time taken by all the delay edge tests performed by delay set analysis.
- *Synch*: Time taken by the dataflow analysis that determines synchronization information.
- *Escape*: Time taken by escape analysis.
- *Barrier*: Time taken by memory barrier insertion to determine where fence or sync instructions are inserted in the code generated.
- *TotalSC*: Total time taken by the just-in-time compiler in the *Delay_SC* configuration, when all our analyses are performed.
- *TotalRC*: Total time taken by the just-in-time compiler in the *Base_Relaxed* configuration, when none of our analysis to compile for SC is performed.

For the first five columns, the number in parentheses is the ratio of the time for the analysis component corresponding to that column over the *TotalRC* time.

The overhead of dynamic compilation necessitates that we use simple analysis techniques. The 3 programs that have much higher overheads are *MolDyn*, *SPECmtrt*, and *SPECjbb00*. Most of the overhead for *MolDyn* comes from memory barrier insertion. *SPECmtrt* and *SPECjbb00* both make use of dynamic class loading, which leads to many non-trivial inter-procedural analysis phases and re-compilations. These contribute to increase the analysis time.

For the Power3 platform, the absolute compile times are higher than the times on the Intel Xeon platform. The base compile time for the *Base_Relaxed* configuration is, in general, also higher.

Table 6 shows the execution time for our programs in the *Base_Relaxed* configuration[2]. Comparing these times with the *TotalSC* time in Table 4 and Table 5, we see that in some cases the total analysis time in our SC compiler overwhelms the execution time of the program when it is run using weak

[2]SPECjbb00 is not included in this table since it measures performance in terms of throughput, and not in terms of execution time.

Benchmark	Delay	Synch	Escape	Barrier	TotalSC	TotalRC
Hashmap	0.1 (0.08)	0.6 (0.46)	1.2 (0.92)	0.9 (0.69)	4.7 (3.61)	1.3
GeneticAlgo	0.1 (0.06)	0.8 (0.44)	1.0 (0.56)	0.7 (0.39)	4.8 (2.67)	1.8
BoundedBuf	0.02 (0.01)	0.3 (0.18)	0.5 (0.29)	0.7 (0.41)	2.9 (1.71)	1.7
Sieve	0.6 (0.10)	0.2 (0.03)	0.3 (0.05)	2.1 (0.37)	8.9 (1.56)	5.7
DiskSched	0.1 (0.08)	0.5 (0.42)	1.9 (1.58)	0.7 (0.58)	4.9 (4.08)	1.2
Raytrace	0.1 (0.04)	0.7 (0.27)	1.5 (0.58)	1.4 (0.54)	6.6 (2.54)	2.6
Montecarlo	0.1 (0.05)	1.1 (0.52)	2.0 (0.95)	0.8 (0.38)	7.2 (3.43)	2.1
MolDyn	0.7 (0.39)	0.6 (0.33)	1.4 (0.78)	14.2 (7.89)	22.6 (12.56)	1.8
SPECmtrt	5.5 (0.83)	9.1 (1.38)	57.9 (8.77)	3.2 (0.48)	89.1 (13.50)	6.6
SPECjbb00	60.1 (1.40)	33.0 (0.77)	340.1 (7.95)	125.1 (2.92)	716.5 (16.74)	42.8

Table 4: Analysis Times in Seconds for the Intel Xeon Platform

Benchmark	Delay	Synch	Escape	Barrier	TotalSC	TotalRC
Hashmap	0.2 (0.05)	2.0 (0.48)	7.9 (1.88)	2.3 (0.55)	18.2 (4.33)	4.2
GeneticAlgo	0.3 (0.06)	2.5 (0.52)	5.7 (1.19)	2.3 (0.48)	16.6 (3.46)	4.8
BoundedBuf	0.03 (0.01)	0.9 (0.22)	1.9 (0.46)	1.9 (0.46)	9.1 (2.22)	4.1
Sieve	0.1 (0.03)	0.8 (0.23)	1.1 (0.32)	1.9 (0.56)	7.9 (2.32)	3.4
DiskSched	0.1 (0.03)	1.7 (0.53)	6.1 (1.91)	1.9 (0.59)	13.9 (4.34)	3.2
Raytrace	0.3 (0.04)	2.3 (0.29)	7.5 (0.94)	4.9 (0.61)	24.3 (3.04)	8.0
Montecarlo	0.3 (0.04)	4.7 (0.60)	9.7 (1.15)	2.7 (0.32)	25.7 (3.06)	8.4
MolDyn	0.4 (0.08)	1.8 (0.37)	5.0 (1.04)	2.6 (0.54)	17.5 (3.65)	4.8
SPECmtrt	12.0 (0.82)	19.4 (1.33)	93.1 (6.38)	11.7 (0.80)	179.7 (12.31)	14.6
SPECjbb00	45.0 (0.94)	31.1 (0.65)	361.2 (7.57)	118.5 (2.48)	716.7 (15.02)	47.7

Table 5: Analysis Times in Seconds for the Power3 Platform

Benchmark	Xeon	Power3
Hashmap	32.8	71.1
GeneticAlgo	13.1	20.7
BoundedBuf	442.6	2131.5
Sieve	168.3	335.4
DiskSched	5.0	3.0
Raytrace	7.6	19.5
Montecarlo	16.0	27.3
MolDyn	5.1	7.6
SPECmtrt	3.4	4.9

Table 6: Execution Time in Seconds for the *Base_Relaxed* Configuration

consistency. However, this is an artifact of the specific problem sizes used in our experiments. As mentioned earlier, we expect the analysis cost to be amortized over the running time in applications that execute for long periods of time.

7. RELATED WORK

In [21], Liblit, Aiken, and Yelick compare the performance difference between SC and the Titanium memory model (a relaxed memory model). They develop a type system to identify shared data accesses in SPMD Titanium programs[32]. For SC, they insert a memory barrier at each shared data access identified. However, in their experiments the benchmark programs chosen show no significant performance degradation when using a naive implementation of SC instead of the relaxed Titanium memory model. This makes it difficult to gauge the impact of their technique. In [31], von Praun uses an object-based approach to simplify delay set analysis. In this approach, the analysis looks for an order satisfied by *all* accesses to a memory location. Our approach is different in that it uses inter-procedural analysis and considers control flow in the program, making it more accurate. For the common benchmarks used, the work in [31] gives a 98.6% performance degradation, whereas the techniques we use give a degradation of 11.5%.

Research has been done to gauge the performance impact of implementing SC in hardware[10, 28]. Experiments with some benchmark programs have shown that speculative techniques can be used to build sequentially consistent architectures with a performance degradation of about 20% on average[13].

Delay set analysis was first described by Shasha and Snir in [29]. Their algorithm is precise, but is shown to be NP-complete in the number of threads[16]. Krishnamurthy and Yelick[15, 16] give a delay set analysis algorithm for SPMD programs that incorporates synchronization information. They use the SPMD property to achieve polynomial time complexity. Their work does not consider thread spawning or termination. Also, their use of locking synchronization to refine the graph considered for delay set analysis differs from our approach. In [5], Chen, Krishnamurthy, and Yelick present an algorithm that uses strongly connected components to perform delay set analysis for SPMD programs. This algorithm has an analysis complexity of $O(n^2)$, where n is the number of shared memory accesses in the program. Previous work in [25, 23, 5] refines delay set analysis to make it more precise for programs that include array accesses.

Callahan and Subhlok[4] give dataflow equations to compute synchronization orders, but their method cannot be generalized to handle recursion. Duesterwald and Soffa[8] use dataflow analysis to compute synchronization orders for a program that may use recursion, and whose code comprises of multiple methods. Their approach determines when one event must happen before and/or after another event. It cannot determine when two events may/must happen con-

currently, and it is unable to differentiate between contexts specific to different call sites for the same method. Masticola and Ryder[22] build on previous work and describe a framework to determine events that may happen in parallel by successively refining the analysis results using various techniques. Naumovich et. al.[26] apply dataflow analysis to compute events that may happen in parallel for programs that use synchronization constructs defined in the Java language, but do not attempt to compute any happens-before or happens-after orders. The synchronization analysis we present in this work also uses dataflow analysis. It computes happens-before, happens-after, and happens-in-parallel relations. It is more precise than previous methods based on dataflow analysis, and can distinguish information at different call sites for the same method.

Pugh has described problems with the semantics of the original memory model for the Java programming language[27], and has led efforts to revise this memory model[14].

8. CONCLUSION

We show that our compiler techniques are effective in reducing the number of delays that need to be enforced when implementing sequential consistency in software. We use an inter-procedural, multithreaded analysis, but a simple, conservative form of it suffices to capture the majority of cases that affect performance. We achieve reasonable performance for a set of benchmark programs compiled assuming SC.

These results indicate that performance may not be a prohibitive factor in the design of language memory models, when considering a model that is stronger than the relaxed models popular today. Use of strong memory models can help the design, development, and maintenance of explicitly parallel programs.

9. REFERENCES

[1] The Java Grande Forum multi-threaded benchmarks. URL: http://www.epcc.ed.ac.uk/javagrande/threads/contents.html.

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

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[4] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, January 1989.

[5] Wei-Yu Chen, Arvind Krishnamurthy, and Katherine Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *Sixteenth Annual Workshop on Languages and Compilers for Parallel Computing*, October 2003.

[6] The Standard Performance Evaluation Corporation. SPEC JVM Client98 suite. 1998. URL: http://www.specbench.org/jvm98/jvm98.

[7] The Standard Performance Evaluation Corporation. SPEC JBB 2000 benchmark. 2000. URL: http://www.specbench.org/jbb2000.

[8] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 36–48. ACM Press, 1991.

[9] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory processing. In *2003 ACM International Conference on Supercomputing*, June 2003.

[10] K. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, 1999.

[11] Stephen Hartley. *Concurrent Programming: the Java Programming Language*. Oxford University Press, 1998.

[12] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[13] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, August 1998.

[14] JavaMemoryModel. Java memory model mailing list. 2003. `www.cs.umd.edu/ pugh/java/memoryModel`.

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

[16] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38:139–144, 1996.

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

[18] Doug Lea. Concurrency utilities package, precursor to Java specification request (JSR) 166: Concurrency utilities. `gee.cs.oswego.edu/dl/concurrency-interest` .

[19] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley, 1999. URL: `http://gee.cs.oswego.edu/dl/cpj`.

[20] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, October 2000.

[21] Ben Liblit, Alex Aiken, and Katherine Yelick. Type systems for distributed data sharing. In *Proceedings of the Tenth International Static Analysis Symposium*, 2003.

[22] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138. ACM Press, 1993.

[23] S. Midkiff. Dependence analysis in parallel loops with i+/-k subscripts. 1995. available as Springer Lecture Notes in Computer Science Vol. N. 1033.

[24] S. Midkiff and D. Padua. Issues in the compile-time optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. II*, pages 105–113, August 1990.

[25] 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.

[26] Gleb Naumovich, George S. Avruninand, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1999.

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

[28] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of The 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 199–210, June 1997.

[29] 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.

[30] Zehra Sura. Analyzing threads for shared memory consistency. October 2004. PhD Thesis (UIUCDCS-R-2004-2475), University of Illinois at Urbana-Champaign.

[31] Christoph von Praun. Efficient computation of communicator variables for programs with unstructured parallelism. In *Seventeenth Annual Workshop on Languages and Compilers for Parallel Computing*, September 2004.

[32] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998.