Automatic Implementation of Programming Language Consistency Models*

Zehra Sura[1], Chi-Leung Wong[1], Xing Fang[2], Jaejin Lee[3], Samuel P. Midkiff[2], and David Padua[1]

[1] University of Illinois at Urbana-Champaign, Urbana, IL 61801
{zsura, cwong1, padua}@cs.uiuc.edu
[2] Purdue University, West Lafayette, IN 47907
{xfang, smidkiff}@purdue.edu
[3] Seoul National University, Seoul 151-742, Korea
jlee@cse.snu.ac.kr

Abstract. Concurrent threads executing on a shared memory system can access the same memory locations. A consistency model defines constraints on the order of these shared memory accesses. For good run-time performance, these constraints must be as few as possible. Programmers who write explicitly parallel programs must take into account the consistency model when reasoning about the behavior of their programs. Also, the consistency model constrains compiler transformations that reorder code. It is not known what consistency models best suit the needs of the programmer, the compiler, and the hardware simultaneously. We are building a compiler infrastructure to study the effect of consistency models on code optimization and run-time performance. The consistency model presented to the user will be a programmable feature independent of the hardware consistency model. The compiler will be used to mask the hardware consistency model from the user by mapping the software consistency model onto the hardware consistency model. When completed, our compiler will be used to prototype consistency models and to measure the relative performance of different consistency models. We present preliminary experimental data for performance of a software implementation of sequential consistency using manual inter-thread analysis.

1 Introduction

A consistency model defines the constraints on the order of accesses to shared memory locations made by concurrently executing threads. For any shared mem-

* This material is based upon work supported by the NSF under Grant No. CCR-0081265, 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 NSF or the IBM Corporation. Also supported in part by the Korean Ministry of Education under the BK21 program and by the Korean Ministry of Science and Technology under the National Research Laboratory program.

B. Pugh and C.-W. Tseng (Eds.): LCPC 2002, LNCS 2481, pp. 172–187, 2005.
© Springer-Verlag Berlin Heidelberg 2005

ory system, it is important to specify a consistency model because that determines the set of possible outcomes for an execution, and enables the programmer to reason about a computation performed by the system.

The simplest and most intuitive consistency model for programmers to understand is sequential consistency (SC). SC requires that all threads "appear to" see the same order of shared memory accesses. Between two consecutive writes to a shared memory location, all threads that access the location see exactly the same value, and that value is consistent with the program order.

Most hardware systems implement consistency models that are weaker than SC [1], i.e. they impose fewer constraints on the order of shared memory accesses. This allows more instruction reordering, thus increasing the potential for instruction level parallelism and better performance. Popular hardware consistency models are weak ordering and release consistency, both of which allow reordering of shared memory accesses. These consistency models assume synchronization primitives that the programmer uses to specify points in the application program where shared memory accesses must be made coherent.

Thus, programmers writing explicitly parallel programs must take into account the consistency model when reasoning about the behavior of their programs. If the consistency model allows indiscriminate reorderings, it makes the task of writing well-synchronized programs difficult. Consider the following code:

```
Thread 1          Thread 2
a = ...;
x = 1;            while (x==0) wait;
                  ...= a;
```

This code uses a busy-wait loop and the variable `x` (initially set to zero) as a flag to ensure that the value of `a` read by Thread 2 is the value that is assigned to `a` by Thread 1. There are no data dependences between statements within a thread. So, for the weak or release consistency model, the access to `a` in Thread 2 may happen *before* the `while` loop has finished execution, or the assignment to `a` in Thread 1 may happen *after* the assignment to `x`. Therefore, there is no guarantee what value of `a` will be read in Thread 2. For these consistency models, the burden is on the programmer to insert proper synchronization constructs to disallow undesirable reorderings.

The ideal consistency model must be simple for programmers to understand. This is especially important because programmers for a general purpose language form a wide user base with varied skill levels.

We are building a compiler infrastructure that uses software to bridge the gap between the hardware consistency model and the consistency model assumed by the programmer. Our compiler is based on the Jikes Research Virtual Machine (Jikes RVM)[1] [2] which is an open-source Java Virtual Machine from IBM. Today, Java is the only widely-used, general-purpose language that defines a consistency model as a part of the language specification [6], i.e. the Java Memory Model. When completed, our compiler will allow users to experiment with alternative consistency models. It is designed to abstract out the effect of

[1] Originally called Jalapeño.

the consistency model on compiler transformations. The compiler will take as inputs a specific consistency model to be assumed by the programmer (henceforth called the software consistency model), and the consistency model provided by the target hardware (i.e. the hardware consistency model). The compiler will perform aggressive inter-thread analysis and optimizations to generate machine code based on the specific hardware and software consistency model(s) chosen.

Many different consistency models have been designed in the past, and new consistency models are still being designed. Since our compiler infrastructure abstracts out the consistency model and parameterizes it, it will allow rapid prototyping of new consistency models. Also, all the analyses and optimizations are to be "truly" portable, i.e. they will be reusable for all hardware and software consistency models we can envision. The availability of a common system platform and compiler algorithms for different consistency models will reduce the number of variables in performance comparison tests. This will allow reliable performance comparisons between different consistency models. Thus, our compiler will serve as a test-bed for developing and evaluating new software consistency models that balance ease of use and performance requirements. Such a test-bed is needed because the programming language community has little experience designing software consistency models. Problems with the Java Memory Model [14] illustrate the difficulty of this task.

We have modified the Jikes RVM to provide a sequentially consistent implementation. SC defines the strongest constraints among popular consistency models that are feasible to implement, i.e. a sequentially consistent implementation has the least flexibility to reorder instructions for optimized execution. Thus, the cost of implementing SC using software gives the maximum performance degradation that can be suffered by any feasible software consistency model implementation on a particular hardware system. We used manual inter-thread analysis to estimate this cost for our compiler using a set of benchmark programs, and found it to be negligible (Section 4). However, a simple implementation using the analyses provided in the Jikes RVM distribution gives slowdowns of 4.6 times on average. Thus, precise inter-thread analysis is important for a software implementation of consistency models that does not sacrifice performance.

The outline of the rest of this paper is as follows. In Section 2, we describe the overall system design for the compiler infrastructure we are developing. In Section 3, we explain the analyses and optimizations that are important from the perspective of implementing different consistency models. In Section 4, we give preliminary experimental data for the performance of a sequentially consistent implementation. Finally, in Section 5, we present our conclusions.

2 System Design

For well-synchronized programs that contain no data races, all shared memory access reorderings possible under a relaxed consistency model (e.g. release consistency) that is weaker than SC are also legitimate under SC. So there is no

inherent reason for the run-time performance of these programs to be different for *any* two consistency models that are equivalent to, or weaker than SC. Thus, it should be possible to provide the user with a consistency model different from the hardware consistency model without suffering performance degradation. The performance of a software implementation of a consistency model is contingent on the compiler's ability to determine, without needing to be overly conservative, all the reorderings that lead to a legitimate outcome. This depends on the precision with which the compiler can analyse the interaction of multiple threads of execution through shared memory accesses. To provide the software consistency model, the compiler inserts *fences* at required points in the generated code. Fences are special hardware instructions that are used for memory synchronization.

To support multiple consistency models, the following features are used:

1. ability to *specify the consistency model* so that it can be parameterized,
2. rigorous *program analysis* for shared memory accesses,
3. constrained *code reordering transformations*, and
4. *fence insertion* to mask the hardware consistency model.

Constrained code reordering transformations and fence insertion have been implemented. They are discussed in detail in Sections 3.5 and 3.6 respectively. Inter-thread analysis algorithms are currently under development, and we discuss some of the issues involved in their design in Section 3.4.

We do not have a suitable notation for specifying the consistency model as yet. The notation we develop must be:

1. easy to use for the person experimenting with different consistency models.
2. easy to translate to a form that can be used by the compiler.
3. expressive, i.e. the design space of new consistency models must not be limited by the ability to specify them in a certain way.

Java uses dynamic compilation, so the choice of a consistency model can be made during program execution, and run-time values can be used to guide this choice. Also, compiler directives may be used to allow different consistency models for code segments within the same program. This flexibility will make it possible for the developer to freely experiment and perhaps gain a better understanding of the complexities involved in the design of software consistency models.

Figure 1 illustrates the relationship between the different components of the compiler. Given the source program and a software consistency model, the program analysis determines the shared memory access orders to enforce. Code reordering transformations use this information to optimize the program without changing any access orders that need to be enforced. Fence insertion also uses this information along with knowledge of the hardware consistency model, and generates code that enforces the access orders required by the software consistency model. It does this by emitting fences for those access orders that are not enforced by the hardware, but are required by the software consistency model.

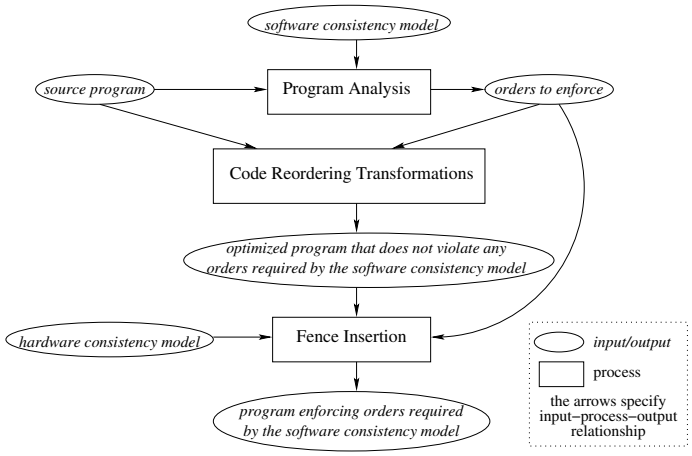

Fig. 1. Components in the design of the compiler

3 Compiler Techniques

3.1 Delay Set Analysis

An execution is invalid when it yields results, visible outside the program, that can only occur if the consistency model is violated. A *delay set* is a set of constraints on program execution order such that if these constraints are enforced, the outcome of the program is always valid according to the consistency model.

In [16], Shasha and Snir show how to find the minimal delay set. They construct a graph where each node represents a program statement. Edges in the graph are of two types: directed program edges (these represent constraints on the order of execution of statements within a thread as determined by the program order), and undirected conflict edges.

A conflict edge exists between two accesses if they are accesses to the same memory location from different threads, they may happen in parallel, and at least one is a write access. Alias analysis and thread-escape analysis (discussed in Section 3.2) help determine if there are accesses to the same memory location from different threads, and MHP and synchronization analysis (discussed in Section 3.3) help determine if they may happen in parallel. The conflict edges can be *oriented* by giving them a direction that represents the order of the two accesses during an execution of the program.

A minimal solution to the delay set analysis problem is the set of program edges in all *minimal mixed* cycles. A mixed cycle is one that contains both program and conflict edges, so a mixed cycle spans at least two threads in the program. Also, since the cycles are minimal, any other cycle in the graph will contain one or more of these minimal cycles within itself. Cycles represent inconsistent executions, because their presence means that there is an orientation of the conflict edges in the graph that is not consistent with the constraints required by the program. The set of program edges in the cycles is the delay set, i.e. the set of constraints that if enforced, prevents an orientation of conflict

edges that gives rise to a cycle. If the program edges in this set are honored, then it is impossible at run-time for an orientation of the conflict edges to occur that gives rise to an inconsistent outcome.

In [9], Krishnamurthy and Yelick show that all exact delay set analysis algorithms for multiple-instruction multiple-data (MIMD) programs have a complexity that is exponential in the number of program segments. For single-program multiple-data (SPMD) programs, a large number of threads can be approximately modeled as two threads, resulting in an exponent of two, and therefore a quadratic algorithm [8]. The algorithms of [8,9] have been implemented, and they give good compile time performance on their set of benchmarks.

In general, programs can have arbitrary control flow and multiple threads may execute different code segments. It is difficult to analyse these programs to accurately determine the conflict edges in the program graph. We will use or develop precise alias analysis, synchronization analysis, and dependence analysis to find the set of conflict edges required for delay set analysis.

We are investigating modified path detection algorithms to find mixed cycles in a program graph. These algorithms have the property that they can be parameterized to be conservative and fast running, or precise and executing in worst case exponential time. They can be adjusted dynamically for different program regions, allowing more precise information to be developed "on demand".

In Java, every thread is an object of class `java.lang.Thread` or its subclasses. Therefore, we can use a type-based approach to approximate the run-time instances of threads.

3.2 Thread Escape Analysis

Escape analysis is used to identify all memory locations that can be accessed by more than one thread. Our analysis is concerned only with thread escaping accesses, and not the more general lifetime or method escaping properties. We are developing an algorithm for this to be as precise as possible. Because our algorithm is specialized for thread escaping accesses, we expect it to be faster than the more general escape analysis algorithms [3,15].

Figure 2 shows a program fragment from *raytracer*[2]. The method `run` creates an instance of the `Interval` object (referenced by `interval`). This object is used only in the statement `render(interval)`, so it can only escape through the call `render(interval)`. In `render`, the value of `interval` is initially accessible only by the thread that assigns this value, and it is never passed to another thread or assigned to static variables. So, `interval` always references a non-escaping object when `render(interval)` is called by `run`, and the fields `interval.width`, `interval.yto`, and `interval.yfrom` are thread local. This information can be derived by a context-sensitive, flow-sensitive escape analysis [3].

Classical optimizations can be applied to uses of `interval` in `render` even if `run` (and hence `render`) is executed by multiple threads. This will not violate the correctness for any consistency model because the object referenced by `interval`

[2] A benchmark program in the Java Grande Forum Multithreaded Benchmarks suite.

```
public void run() {
   Interval interval = new Interval(...);
   ...
   render(interval);
   /* interval is not used hereafter */
   ...
}

public void render(Interval interval) {
   int row[] = new int[interval.width * (interval.yto - interval.yfrom)];
   ...
   for (y = interval.yfrom+interval.threadid; y < interval.yto;
            y += JGFRayTracerBench.nthreads) {
      ylen = (double) (2.0*y) / (double) interval.width - 1.0;
      ...
   }
   ...
}
```

Fig. 2. Example program to illustrate thread local objects

```
public void render(Interval interval) {
   int width = interval.width;
   int yto = interval.yto;
   int yfrom = interval.yfrom;
   int row[] = new int[width * (yto - yfrom)];
   ...
   for (y = yfrom+interval.threadid; y < yto;
                  y += JGFRayTracerBench.nthreads) {
      ylen = (double) (2.0*y) / (double) width - 1.0;
      ...
   }
...
}
```

Fig. 3. Optimization using information about thread local objects

does not escape the thread it was created in. Thus, redundant loads can be removed from `render` because no other thread can access the object referred by `interval`. The optimized `render` method is shown in Figure 3.

3.3 MHP and Synchronization Analysis

May-happen-in-parallel (MHP) analysis determines which statements may execute in parallel. We are developing an efficient (low order polynomial time) algorithm to perform MHP analysis that builds on previous work [13]. We use a program thread structure graph, similar to a call graph, that allows parallelism between code regions to be easily determined.

Synchronization analysis [4] is used to refine the results of MHP analysis. For two statements *S1* and *S2* that are in different threads and may happen in parallel, it attempts to determine if *S1* must execute before *S2*, or after *S2*. This can reduce the number of possible executions, and improve the precision of other analyses. For example, concurrent global value numbering [12] can become more precise since the values assigned to a variable in one thread that can reach a use of that variable in another thread can be determined more accurately. Also, the precision of delay set analysis improves because synchronization analysis can determine orders on pairs of shared memory accesses from different threads that would otherwise be assumed to conflict.

3.4 Issues in the Design of Analysis Algorithms

To gauge the true effect of a consistency model on performance, the compiler must use detailed inter-thread analysis. The analyses and optimizations must be precise, otherwise this may lead to missed opportunities for code reordering optimizations. This will adversely affect software consistency models that are stronger than others being tested. The performance for the strong consistency models will not reflect their true potential, and so fair comparisons between different consistency models will not be possible.

We will be implementing alias analysis, concurrent global value numbering (CGVN), thread-escape analysis, MHP analysis, synchronization analysis, and delay-set analysis. There are several issues that make it challenging to design analysis algorithms for our compiler:

Run-time compilation: Faster compilation is always desirable, but for a system where the compile time contributes to the run-time, speed of compilation is imperative. Our system uses dynamic compilation, so it is important that it does not spend too much time performing analyses and optimizations. Thus, the algorithms we use must be *fast*.
Dynamic class loading: Due to Java's dynamic class loading feature, it is possible that a method being analyzed contains calls to other unresolved methods. This can result in incomplete information for analysis. Thus, we plan to implement algorithms that are designed to be *incremental*. A set of classes that are inter-dependent or mostly used together can be packaged into an archive (for example, as a JAR file). For these classes, inter-procedural and inter-class analysis information that is independent of the dynamic context can be determined statically and included in the package. Later, when a class from this package is first used in an application, the archived analysis results will be readily available for the dynamic compiler to incorporate into its current context. This helps to improve both the speed and accuracy of the analysis. Ideas from the design of the NetBeans [19] and Eclipse [18] platforms can be used to integrate a static package that provides analysis information into a dynamic execution at run-time.
Incomplete program information: Dynamic class loading and JIT compilation require that the analyses and optimizations be done with incomplete program information. E.g., when an object is passed to an unresolved method, escape analysis cannot determine if the object is thread-escaping since it does not know whether the method makes the object accessible to another thread. Thus, a lack of information leads to conservative and imprecise analysis, which can degrade performance. There are two ways to handle incomplete program information:

1. The analysis can be *optimistic* and assume the best-case result. Thus, when faced with lack of information, escape analysis would assume an object is non-escaping. The compiler will then perform checks when methods are compiled later to ensure that previous assumptions are not violated. If a violation occurs, it triggers a recompilation to patch up code previously generated.
2. The analysis makes conservative assumptions when faced with lack of information, but always generates safe code. Later, when more information

becomes available, methods can be *recompiled* and more precise results obtained. To avoid expensive re-analysis overhead, only execution hotspots will be recompiled. These hotspots typically include code sections with many fences because fences are expensive. Recompiling the code using more analysis information allows greater freedom to reorder code and may eliminate some fences. The cost model for recompiling must take into account both the number of fences in the hotspot, and the potential benefit in performance of the transformed code after recompilation.

Both approaches require an adaptive recompilation system that allows methods to be recompiled during execution of the program.

3.5 Reordering Transformations

Code reordering transformations are sensitive to the consistency model being supported because they can change the order of accesses to shared memory locations. Therefore, optimization algorithms developed for single-threaded sequential programs that use code motion or code elimination cannot be directly applied to shared memory systems with multiple threads of execution.

Consider the following example, where `r1`, `r2`, and `r3` are registers, and `a` and `b` are aliased.

```
1.  Load r1, [a]  // "a" contains the address of a memory location to load from
2.  Load r2, [b]  // "b" contains the same address as "a"
3.  Load r3, [a]
```

If redundant load elimination (RLE) is applied to this code segment, the third instruction can be transformed into `Move r3, r1`. However, this transformation will violate SC if there is another thread that executes between the first and second `Load`, and changes the value of the memory location given by `a`. In this case, `r3` gets an older value than `r2`, even though the assignment to `r3` occurs after the assignment to `r2`. Thus, the RLE transformation appears to reorder code by changing the order of statements 2 and 3.

Transformations must be inhibited if they reorder code in cases where delay set analysis determines that the software consistency model may be violated. For best performance, there must be few instances where a transformation is inhibited.

Our initial implementation targets SC as the software consistency model. We examine all transformations in our compiler to determine if they are valid for a sequentially consistent execution. Our compiler is built using the Jikes RVM, so we focus on optimizations implemented in the Jikes RVM. We modify transformations so that they are not performed for instances that can potentially violate the software consistency model; however, these transformations are performed, even in sequentially consistent programs, where they are shown to be safe by the analyses previously described. Optimizations in the Jikes RVM that are affected are redundant load elimination, redundant store elimination, loop invariant code motion, scalar replacement of loads, and loop unrolling.

In the Jikes RVM, optimization phases are structured such that redundant load/store elimination is done before most other optimizations. Thereafter, the number of accesses to memory variables are fixed in the code. Further optimizations work on the results of these load/store instructions and temporary variables. When tailoring optimization phases to account for a particular memory model, the effects of code motion on the relative ordering of load/store instructions have to be accounted for. However, *elimination* of loads and stores is not a cause for concern at this stage if only temporary variables are being eliminated. For example, common subexpression elimination (CSE) done across basic blocks, as currently implemented in the Jikes RVM, eliminates common subexpressions involving only temporary variables that cannot be changed by another thread. So this CSE does not need to be modified for different consistency models.

In previous work [10,12], we give algorithms to perform optimizations such as constant propagation, copy propagation, redundant load/store elimination, and dead code elimination for explicitly parallel programs. These algorithms are based on concurrent static single assignment (CSSA) graphs. CSSA graphs are constructed from concurrent control flow graphs and they preserve the SSA property for parallel programs, i.e. all uses of a variable are reached by exactly one static assignment to the variable. CSSA graphs and delay set analysis enable us to encapsulate the effects of the consistency model. Thus, the algorithms described can be reused for different consistency models by supplying the consistency model as a parameter to the optimization engine. We will apply these techniques in the compiler we are developing.

Example. We illustrate the use of inter-thread analysis to modify transformations in our compiler: we show how delay-set analysis influences RLE using the toy example in Figure 4. This example is based on a code segment extracted from *lufact*[3] and simplified for clarity. We assume that the software memory model is SC, and that the execution uses only two threads.

```
Thread 0                          Thread 1
...                               ...
P1: while (!sync.flag1) ;         S1: ...= col_k[l];
P2: col_k[l] = ...;               S2: sync.flag1 = true;
P3: sync.flag2 = true;            S3: while (!sync.flag2) ;
                                  S4: ...= col_k[l];
...                               ...
```

Fig. 4. Code segment to illustrate delay-set analysis

For the example, global value analysis and alias analysis are used to determine that all instances of `col_k[l]` refer to the same element of a single array object. The benchmark program *lufact* performs Gaussian elimination with partial pivoting for a matrix. The rows of the matrix are distributed for processing amongst different threads. However, `Thread 0` is responsible for assigning the pivot element (`col_k[l]` in the example) each time. It uses shared variables `sync.flag1` and `sync.flag2` for synchronization to avoid conflicts (the benchmark program uses volatile variables to implement barriers for this purpose).

[3] A benchmark program in the Java Grande Forum Multithreaded Benchmarks suite.

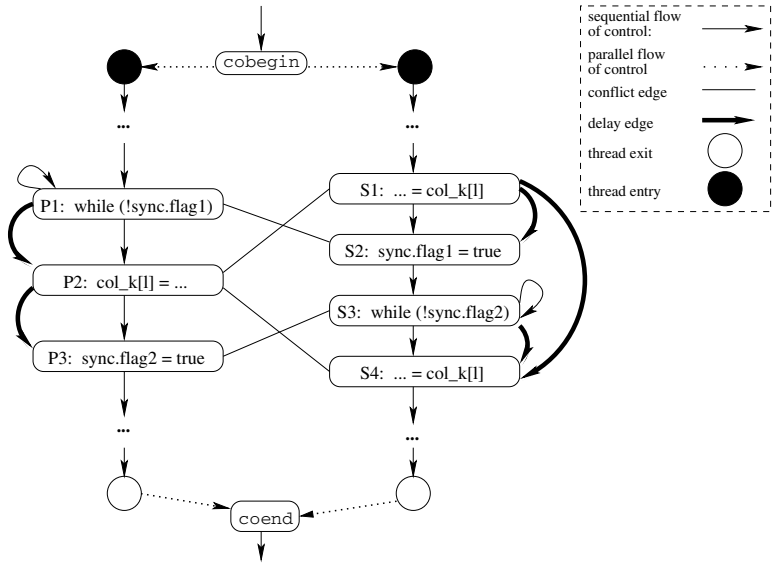

Fig. 5. Program graph for the example of Figure 4 showing delay edges

The program graph with delay edges is shown in Figure 5. There are conflict edges between P1 and S2 for the synchronization using `sync.flag1`, between P3 and S3 for the synchronization using `sync.flag2`, between P2 and S1 for the write-read access to `col_k[l]`, and between P2 and S4 for the write-read access to `col_k[l]`. These result in the following three cycles in the program graph:

1. P1,P2,S1,S2: the corresponding delay edges are P1-P2 and S1-S2.
2. P2, P3, S3, S4: the corresponding delay edges are P2-P3 and S3-S4.
3. P2, S1,S4: the corresponding delay edge is S1-S4.

The delay edges are enforced by inserting fences as discussed in Section 3.6.

If we consider each thread in isolation without any consistency model constraints, RLE may be applied to the access `col_k[l]` at S4. Thus, by traditional analysis, the value of `col_k[l]` accessed at S1 may be stored in a register and reused at S4. However, this is invalid for SC because it effectively reorders the access at S3 and the access at S4. Inter-thread delay set analysis can recognize this, and the delay edge from S3 to S4 prevents the reuse of the value accessed at S1.

Note that the synchronization is important to guarantee consistent execution, and it is the effect of the synchronization variable `sync.flag2` that ensures the RLE optimization is not incorrectly applied in this instance. For SC, the presence of an intervening conflicting access is needed to disallow the reordering of two conflicting accesses to the same shared variable. Taking this fact into account helps to perform optimizations in as many instances as possible without being overly conservative.

3.6 Fence Insertion

Fence instructions force memory accesses that were issued previously to complete before further processing can occur. They can be used to enforce necessary orders

for a consistency model, and thus it is possible to mask the hardware consistency model by another stricter[4] consistency model [11]. Thread-based escape analysis, MHP analysis, synchronization analysis, and delay-set analysis determine the points in the code where fences must be inserted.

We describe two techniques that optimize the number of fences inserted. First, register hazards (i.e. write/write, write/read, and read/write operations to the same register) in conjunction with the store order from the processor enforce an order for all accesses to a particular memory location that occur within the same thread. This is exploited to reduce the number of fences that are inserted. Second, two orders to be enforced can overlap, i.e. in the sequence of operations:

```
load A;     load B;     load C;
```

it is possible that the orders to be enforced are A before C and B before C. Fences can be inserted in at least two ways: by placing a fence immediately after A and immediately after B, or, more efficiently, by placing a single fence after B that enforces both of the orders.

When developing an algorithm that makes the best use of program control flow to optimize fence insertion, additional complexity is added for programs with loops and branches. We have implemented a fast, flow-insensitive algorithm, and a slower, but more precise, flow sensitive form of the analysis [5]. The experimental data in Section 4 was obtained using the whole-procedure, flow-sensitive algorithm for fence insertion.

4 Experiments

We are implementing the compiler infrastructure described using the Jikes RVM on an IBM AIX/PowerPC platform. The IBM PowerPC supports a relaxed consistency model that is similar to weak ordering and provides *sync* instructions that can be used as fences. The Jikes RVM is a Java virtual machine written mostly in Java. It does dynamic compilation for the whole program, i.e. before any method is executed, machine code is generated for the entire method. This allows fences to be inserted to enforce the software consistency model. The Jikes RVM also supports an adaptive recompilation system that can be used to optimize in the presence of dynamic class loading.

We obtained preliminary experimental results for a software implementation of SC using both simple analysis and manual inter-thread analysis. We describe simple analysis in Section 4.1 and manual inter-thread analysis in Section 4.2. The goal of the experiments is to obtain quantitative numbers to estimate the performance of an implementation of the compiler we have described. As a first step, we focus on using SC as the programming language consistency model. This is because SC is a natural and intuitive programming model and is one of the most expensive consistency models to implement.

[4] A stricter consistency model specifies stronger constraints on the ordering of shared memory accesses.

For this study, we choose benchmark programs that have a small code size, because that makes them amenable for manual analysis to be applied. We use well-synchronized Java programs because these programs introduce no more constraints for a sequentially consistent programming model than a relaxed consistency model. The synchronization required for these programs is the same for SC as for popular hardware consistency models. We rely on program analyses to detect the set of "required" synchronizations, i.e. the set of shared memory access orders that must be enforced in program threads.

4.1 Simple Analysis

The simple analysis is the default analysis that is provided in the Jikes RVM distribution. There is no MHP, synchronization, or delay set analysis. The escape analysis is a simple flow-insensitive analysis. Initially, all variables of reference type are assumed not to be pointing to any thread-shared objects. The analysis then marks a reference variable as pointing to a thread-shared object if:

1. it stores the result of a load from a memory location, or
2. it is the operand of a store to a memory location, or
3. it is the operand of a return statement (and not a method parameter), or
4. it is the operand of a throw statement, or a reference move statement.

The performance using this conservative simple analysis provides a lower bound of the performance using automatic inter-thread analysis. The simple analysis may conservatively mark arguments of method calls as pointing to thread-shared objects, even if the object is never passed to another thread. Therefore, using this analysis can degrade performance even when the program is single-threaded.

4.2 Manual Inter-thread Analysis

The analysis results are statically determined and supplied to the compiler by inserting special directives in the program. These directives mark the accesses to thread-shared memory objects that need to be synchronized. Although the directives are inserted manually, we have the compiler implementation in mind when we insert them. Logically, we mark an access if:

- the access involves a thread escaping object, i.e. the object is assigned to or obtained from a class field, or passed to or obtained from another thread.
- MHP and delay set analysis cannot prove the thread escaping object is accessed exclusively.

Therefore, we expect the performance using manual inter-thread analysis to be the upper bound of the performance using automatic inter-thread analysis. Note that it is an upper bound because we did not consider dynamic class loading in manual analysis. In the automatic analysis, we may only have partial program information due to unloaded classes.

4.3 Experimental Data

Table 1 gives execution times obtained for the benchmark programs. The first seven programs are from the SPECJVM98 Benchmark suite, the next two are from ETH, Zurich[5], and the remaining are from the Java Grande Forum Benchmark suite v1.0. Except for the first six, all the programs are multithreaded.

Our goal is not to provide higher performance, but comparable performance with an easy-to-understand-and-use consistency model. We compare the slowdown when running the benchmark programs with SC, versus using the default consistency model implemented in the Jikes RVM.

The data presented in Table 1 gives the average times taken over 4 runs of each program on an IBM SP machine with 8 GB of memory and using 4 375MHz processors. The 'Original' column is the execution time for the program using the default Jikes RVM Java memory model. Column 'Simple' is the time taken for an implementation of sequential consistency that uses the simple analysis previously described. Column 'Manual' is the time taken for an implementation of sequential consistency that uses precise, manual, inter-thread analysis. Note that the slowdown numbers are shown in parentheses.

Table 1. Performance, in seconds, with (i) simple escape analysis, and (ii) manual escape and delay-set analysis

Benchmark	Original	Simple	Manual
_201_compress	16.812	224.924 (13.379)	17.038 (1.013)
_202_jess	9.758	29.329 (3.006)	9.777 (1.002)
_209_db	30.205	41.232 (1.365)	30.560 (1.011)
_213_javac	15.964	36.706 (2.299)	15.606 (0.978)
_222_mpegaudio	13.585	174.478 (12.843)	13.457 (0.991)
_228_jack	19.371	41.070 (2.120)	19.326 (0.998)
_227_mtrt	4.813	26.516 (5.509)	4.822 (1.002)
elevator	22.507	22.509 (1.000)	22.508 (1.000)
philo	15.391	15.817 (1.028)	15.465 (1.004)
crypt	23.579	32.688 (1.386)	23.563 (0.999)
lufact	3.177	3.514 (1.106)	3.182 (1.001)
series	141.859	378.768 (2.670)	141.065 (0.994)
sor	4.137	35.776 (8.648)	4.137 (1.000)
sparsematmult	3.788	26.062 (6.880)	7.499 (1.979)
moldyn	71.756	446.406 (6.221)	71.190 (0.992)
montecarlo	13.973	28.852 (2.065)	13.846 (0.991)
raytracer	145.145	1015.980 (7.000)	145.237 (1.001)

For simple escape analysis, the average slowdown was 4.6, and 12 of the 17 benchmarks showed slowdowns greater than 2 times. However, for manual analysis, none of the benchmarks showed any significant slowdowns, except *sparsematmult*. *sparsematmult* showed a slowdown of 2 times. It performed poorly because the analysis techniques we have described are not sufficient to eliminate false dependences in the program that occur due to indirect array indexing. However, other techniques can be applied to reduce the slowdown for *sparsematmult*. For example, speculative execution or run-time dependence analysis can be used.

[5] Thanks to Christoph von Praun for these.

Our results show that for well-synchronized programs, it is feasible to implement consistency models in software without performance degradation. So, it is worthwhile to develop precise and efficient analysis algorithms for this purpose.

5 Conclusion

We have outlined the design of a compiler that allows the consistency model to be a programmable feature. This compiler can be used to prototype and test new consistency models, and to determine the effect of a consistency model on compiler transformations and run-time performance.

We obtained an estimate for performance degradation due to a software implementation of sequential consistency that uses manual analysis. For all benchmark programs except one, there was no slowdown. This demonstrates the usability of software consistency model implementations to provide ease-of-use to programmers, and justifies the development of precise and efficient inter-thread analysis algorithms for this purpose.

References

1. Sarita V. Adve and Kourosh Gharachorloo.: Shared memory consistency models: A tutorial. IEEE Computer, pages 66-76, Dec 1996
2. B. Alpern, et al: The Jalapeno virtual machine. IBM Systems Journal, Feb 2000
3. J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff: Escape analysis for Java. Proceedings ACM 1999 Conference on Object-Oriented Programming Systems (OOPSLA 99), pages 1-19, Nov 1999
4. P.A. Emrath, S. Ghosh, and D.A. Padua: Event synchronization analysis for debugging parallel programs. Proceedings of Supercomputing, pages 580-588, 1989
5. Xing Fang: Inserting fences to guarantee sequential consistency. Master's thesis, Michigan State University, July 2002
6. J. Gosling, B. Joy, G. Steele, and G. Bracha: The Java Language Specification, Second Edition. The Java Series, Addison-Wesley Publishing Company, Redwood City, CA 94065, USA, 2000
7. Mark D. Hill: Multiprocessors should support simple memory-consistency models. IEEE Computer, pages 28-34, Aug 1998
8. Arvind Krishnamurthy and Katherine Yelick: Optimizing parallel SPMD programs. Seventh Workshop on Languages and Compilers for Parallel Computing, Aug 1994
9. Arvind Krishnamurthy and Katherine Yelick: Analyses and optimizations for shared address space programs. Journal of Parallel and Distributed Computing, 38:139-144, 1996
10. J. Lee, S.P. Midkiff, and D.A. Padua: Concurrent static single assignment form and constant propagation for explicitly parallel programs. Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing, pages 114-130, Springer, Aug 1997
11. J. Lee and D.A.Padua: Hiding relaxed memory consistency with compilers. Proceedings of The 2000 International Conference on Parallel Architectures and Compilation Techniques, Oct 2000

12. J. Lee, D.A.Padua, and S.P. Midkiff: Basic compiler algorithms for parallel programs. Proceedings of The 1999 ACM SIGPLAN Symposiun on Principles and Practice of Parallel Programming, pages 1-12, May 1999
13. G. Naumovich, G.S. Avruninand, and L.A. Clarke: An efficient algorithm for computing MHP information for concurrent Java programs. Proceedings of Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sep 1999
14. William Pugh: Fixing the Java memory model. Proceedings of the ACM 1999 Java Grande Conference, June 1999
15. R. Rugina and M. Rinard: Pointer analysis for multithreaded programs. Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pages 77-90, June 1999
16. 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, Apr 1988
17. C.-L. Wong, Z. Sura, X. Fang, S.P. Midkiff, J. Lee, and D. Padua: The Pensieve Project: A Compiler Infrastructure for Memory Models. The International Symposium on Parallel Architectures, Algorithms, and Networks, May 2002
18. Eclipse Platform Technical Overview. Object Technology International, Inc., July 2001, Available at www.eclipse.org
19. The NetBeans Platform. Sun Microsystems, Inc., Documentation available at www.netbeans.org