Exploiting Task Order Information for Optimizing Sequentially Consistent Java Programs

Christoph M. Angerer, Thomas R. Gross
ETH Zurich, Switzerland
angererc@inf.ethz.ch, trg@inf.ethz.ch

***Abstract*—Java was designed as a secure language that supports running untrusted code as part of trusted applications. For safety reasons, Java therefore defines a memory model that prevents undefined behavior in multi-threaded programs even if the programs are not correctly synchronized.**

Because of the potential negative performance impact the Java designers did not choose a simple and natural memory model, such as sequential consistency, but instead developed a relaxed memory model that gives the compiler more optimization opportunities. As it is today, however, the relaxed Java Memory Model is not only hard to understand but it unnecessarily complicates reasoning about parallel programs and it turned out to be difficult to implement correctly.

This paper presents an optimizing compiler for a Java version that has sequential consistency as its memory model. Based on a programming model with explicit happens-before constraints between tasks, we describe a static schedule analysis that computes whether two tasks may be executed in parallel or if they are ordered. During optimization, the task-ordering information is exploited to reduce the number of volatile memory accesses the compiler must insert to guarantee sequential consistency.

The evaluation shows that scheduling information significantly improves the effectiveness of the optimizations. For our set of multi-threaded benchmarks the fully optimizing compiler removes between 70% and 100% of the volatile memory accesses inserted by the non-optimizing compiler. As a result, the overhead of sequentially consistent Java compared to standard Java is reduced from 136% on average for the unoptimized version to 11% on average for the optimized version. The results indicate that with appropriate optimizations, sequential consistency can be a feasible alternative to the Java Memory Model.

I. Introduction

From the beginning, Java was designed to be a safe and secure language that supports running untrusted code as part of trusted applications. As a multi-threaded language, Java therefore must prevent undefined behavior even for programs that are not correctly synchronized. The *Java Memory Model* [1] formally defines the basic semantics of shared variables and specifies what values a read of a shared memory location is allowed to return.

For correctly synchronized code, that is, programs without data races, the Java Memory Model guarantees *sequential consistency*. In a sequentially consistent system, all threads see write operations to the same memory location in the same order and the operations of each individual thread appear in the order specified by its program text. Because each thread must behave according to its program text, it is relatively easy for programmers to reason about sequentially consistent

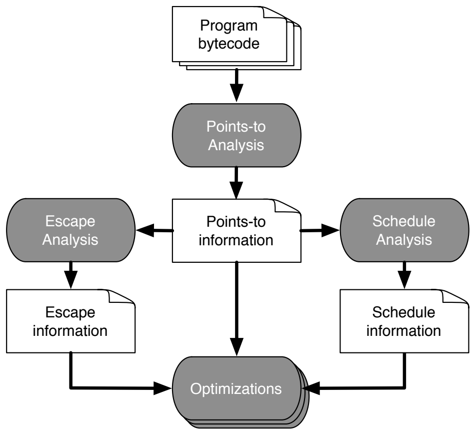

Fig. 1. Dependencies between the different analyses and optimizations.

programs. However, many hardware and compiler optimizations commonly used in uniprocessors, such as register promotion, common subexpression elimination, and reordering, can potentially violate sequential consistency: the effects of an optimization in one thread may be observed by another thread and the program-text order can appear to be violated.

The Java Memory Model tries to minimize the possibly negative performance impact of strict memory models by relaxing the constraints on the order of shared memory accesses. This increases opportunities for the compiler to apply well-known uniprocessor optimizations. However, a relaxed memory model not only makes it harder for programmers to reason about their applications; it also turned out to be extremely difficult to implement correctly [2].

This paper presents an optimizing compiler for a Java version with a sequentially consistent memory model. Because the compiler is a bytecode-to-bytecode translator, the programs can run on an unmodified standard Java Virtual Machine. The compiler exploits task-ordering information gathered from the program to compute whether two tasks are potentially executed in parallel or whether they are ordered by happens-before relationships. This ordering information helps the compiler to decide for each program point whether a change in this point may be observed by a parallel task (and thus violate sequential consistency) or not.

II. An Optimizing Compiler for Sequentially Consistent Java

Figure 1 gives an overview of the optimizing compiler and its three most important analyses: *points-to analysis*, *escape analysis*, and *schedule analysis*. This section describes the points-to analysis and escape analysis as well as the optimization component. The schedule analysis is the main contribution of this paper and described in more detail in Section IV.

At the core of the optimization for sequentially consistent Java is the problem of finding interfering program points. In a sequentially consistent program, an optimization can be applied at a program point P if there is no other program point Q that at runtime may be executed in parallel and that may observe the change in P[1]. In general, two program points P and Q are said to interfere at runtime if they fulfill the following two criteria:

1) The two program points P and Q access the same runtime object Obj_{rt}. At compile-time, this can be approximated by a *points-to analysis* that checks whether P and Q may point to the same analysis object Obj. By adding an *escape analysis* that computes whether Obj is local to a task $T()$ the number of interfering program points can be further reduced.
2) P and Q are executed in parallel. Information about what tasks may be executed in parallel is computed at compile-time by a *schedule analysis*.

The compiler is realized as a bytecode-to-bytecode translator. It rewrites all memory accesses at interfering program points (accesses to object fields as well as accesses to elements inside an array) into calls to `sun.misc.Unsafe.getXYZVolatile()` and `sun.misc.Unsafe.putXYZVolatile()`[2]. The volatile getters and setters in the class `Unsafe` guarantee that the memory accesses follow the Java **`volatile`** semantics thus creating happens-before relationships between concurrent reads and writes.

The goal of the optimizations is to reduce the number of such volatile memory accesses. This not only removes the overhead directly associated with a volatile memory access but also increases the optimization opportunities for the just-in-time compiler when the program is executed.

A. Points-to Analysis and Escape Analysis

Starting from a bytecode representation of the program, the points-to analysis computes points-to sets for each program variable. Virtual call sites are also resolved by this phase, using the points-to information as well as type information to find the possible target methods for a call site.

The points-to analysis can be used to decide whether two program points P and Q may interfere in a parallel program by comparing the points-to sets at P and Q. If the points-to sets are disjoint, then the sets of runtime objects accessed at P and Q will be disjoint too, and so the program points do not interfere. If the points-to sets overlap, however, there may be runtime objects that will be accessed at P and Q concurrently.

MAY-INTERFERE

$$\frac{\begin{array}{c}(1a)\ P\ \texttt{accesses}\ \mathit{v1}\ \texttt{in}\ A()\\ (1b)\ Q\ \texttt{writes}\ \mathit{v2}\ \texttt{in}\ B()\\ (2a)\ \mathit{v1}\ \texttt{may point to}\ \mathit{Obj}\\ (2b)\ \mathit{v2}\ \texttt{may point to}\ \mathit{Obj}\\ (3)\ \mathit{Obj}\ \texttt{may escape}\ A()\ \texttt{and}\ B()\\ (4)\ \mathit{parallel}(A(),\ B())\end{array}}{\mathit{mayInterfere}(P,\ Q)}$$

Fig. 2. The basic optimization question.

The escape analysis is based on the points-to analysis and uses the points-to sets to find all task objects that may be created during the execution of a program. For each task object $T()$, the escape analysis starts at the task entry method of $T()$ and follows the resolved call-graph to find all methods $m()$ that may be invoked during the execution of $T()$. For each object Obj created in a method $m()$ reachable by $T()$ the escape analysis decides whether Obj may escape $T()$. An object Obj is said to escape $T()$ if it is passed as a parameter to $T()$'s entry method, if it may be stored in a static field, if it may be passed as a parameter to another task, or if it is reachable by any object that escapes $T()$.

It is interesting to note that neither the points-to analysis nor the escape analysis consider task ordering at all. This additional information is provided by the schedule analysis presented in this paper.

B. Optimziations

Figure 2 shows the MAY-INTERFERE optimization rule. MAY-INTERFERE computes all pairs of program points P and Q with potential read/write or write/write interference.

The result of the MAY-INTERFERE rule is a relation *mayInterfere* between pairs of program points P and Q. For program points P and Q that are reachable by tasks $A()$ and $B()$ respectively, clauses $(1a)$ and $(1b)$ select the variables $v1$ and $v2$ that P and Q access. Clauses $(2a)$ and $(2b)$ consult the points-to sets to check whether $v1$ and $v2$ may point to the same analysis-time object Obj. The third clause tests whether Obj may escape either task $A()$ or $B()$. If in clause (4) the schedule analysis further found that $A()$ and $B()$ may be executed in parallel, the optimization concludes that program points P and Q may interfere with each other at runtime and thus the memory access must be made volatile.

The optimization for reducing the number of volatile memory accesses described here can be complemented by additional optimizations that make use of scheduling information. [4], for example, presents optimizations such as synchronization removal or removal of read/write barriers in software transactional memory systems that can be integrated with our compiler. The resulting compiler optimizes different aspects of multi-threaded programs that are not necessarily all related to the sequentially consistent memory model.

[1]This is a simple but conservative approximation of the critical cycle detection in a delay-set analysis [3].

[2]`XYZ` is a placeholder for the various basic Java types such as `int`, `float`, or `Object`.

III. A Task Model with Explicit Task Ordering Constraints

The optimizations presented in this paper exploit the fact that many tasks in a parallel program do not actually execute in parallel with each other but are explicitly or implicitly ordered. Imagine, for example, a program that executes in multiple distinct phases. While tasks within one phase may run in parallel, the phase structure ensures that two tasks from different phases are ordered and therefore can never participate in a data race.

Many modern parallel programming libraries and languages allow the programmer to specify task orderings with varying degrees of explicitness. Structured fork-join style systems, for example, syntactically enforce that the forking and joining task is ordered with respect to the forked sub-tasks. Other systems, such as Apple's Grand Central Dispatch or Microsoft's Task Parallel Library, allow the programmer to explicitly define happens-before relationships between task objects thus avoiding the limitations of lexically scoped parallelism.

This section describes a model for fine-grained parallelism based on lightweight tasks with explicit happens-before relationships. We base our model on the work of Angerer et al. [5] because this model is general enough to express a wide variety of existing concurrency patterns: structured fork-join style parallelism, semi-structured tasks with ordering information, and unstructured threads.

The basic building block of the execution model is a *task*. A task is similar to a method in that it contains code that is executed in the context of a **`this`**-object (or the class, in the case of **`static`** methods/tasks). Unlike a method, however, one does not *call* a task, which would result in the immediate execution of the body, but instead *schedules* it for later execution.

As an example, consider a task `T()` that starts two long-running computations `Compute1()` and `Compute2()` and schedules a task `Print()` that will print the result after the computations have finished:

```
task T() {
  Task print = schedule this.Print();
  Task compute1 = schedule this.Compute1();
  Task compute2 = schedule this.Compute2();
  compute1→print;
  compute2→print;
}
```

A task schedule is represented as a graph of `object.Task()` pairs. The statement **`schedule this`**`.Print()`, for example, creates a new node in the schedule with the **`this`** object and the `Print()` task method and returns an object of type `Task` representing that node. Like any other object, `Task` objects can be kept in local variables, passed around as parameters, and stored in fields.

At runtime, a scheduler constantly chooses tasks that are eligible for execution and starts them. The order in which the scheduler is allowed to start the tasks is specified by the edges in the schedule graph. In the example, the statement `compute1→print` creates an explicit happens-before relationship between the two referenced task objects and adds the happens-before edge **`this`**`.Compute1()→`**`this`**`.Print()` to the schedule. In the above program, the scheduler guarantees that both tasks `Compute1()` and `Compute2()` have finished execution before task `Print()` is started. The tasks `Compute1()` and `Compute2()`, however, are not ordered and may therefore be executed in parallel.

Whenever a new task is scheduled, the scheduler automatically adds an initial happens-before relationship between the currently executing task and the newly created task node. These implicit edges prevent the immediate execution of the newly scheduled tasks and enable the current task to add additional constraints to the schedule before it finishes.[3].

A task method can also take task objects as parameters. One common use for task parameters is to allow task methods to schedule subtasks relative to tasks that have been scheduled outside. In the next example, `Compute()` may use additional subtasks to perform its computation. In this case, all subtasks must have finished before the `Print()` task executes. To implement this behavior, we can pass the `Print()` task object as a parameter to `Compute()` and schedule the subtasks before it:

```
task T() {
  Task print = schedule this.Print();
  //pass reference to print task:
  Task compute = schedule this.Compute(print);
  compute→print;
}
task Compute(Task later) {
  Task subtask = schedule this.Subtask(later);
  //schedule subtask before later
  subtask→later;
}
```

In `Compute()` we pass the reference `later` even further along to `Subtask()`, thus allowing `Subtask()` (and its subtasks, if there are any) to push the execution of the `Print()` task further and further into the future until the whole computation has finished. Once the subtasks terminate without inserting new tasks, the scheduler will be able to execute `Print()`.

A. Example with Threads and Barriers

Figure 3 shows a simplified excerpt of the `sor` benchmark (successive over-relaxation over a 2D grid). In method `begin()`, `sor` first starts a number of worker threads on line 7 before the main thread waits for all workers to finish on line 14 in method `end()`.

Every worker successively refines the overall result by repeating the same computation a number of times. Each worker executes the same loop on line 25. As a data-parallel application, the workers generally work in parallel on separate parts of the input data (not shown in the example). Due to data-dependences, however, all workers must synchronize on a barrier twice in every iteration on lines 27 and 29 to separate the two computation phases on lines 26 and 28.

[3]This behavior—delaying sub-tasks until the parent task has finished—has been chosen to keep the model as small and general as possible. The effect of an explicit `start()` method can be achieved by a program transformation similar to *continuation passing style*: at each call to `start()` the parent task is split into two tasks, one for the prelude and one for the remainder of the task [5].

```
public class Sor {
  Barrier barrier = new Barrier(NumWorkers);
  Worker[] workers = new Worker[NumWorkers];
  void begin() {
    for(int i=0; i<NumWorkers; i++) {
      workers[i] = new Worker(this, i);
      workers[i].start();
    }
    this.end();
  }
  void end() {
    for(int i=0; i<NumWorkers; i++) {
      try {
        workers[i].join();
      } catch(InterruptedException e) {}
}}}

class Worker extends Thread {
  Sor sor; int id;
  Worker(Sor sor, int id) {
    this.sor = sor;
    this.id = id;
  }
  void run() {
    for(int count=0; count<NumRounds; count++)
      /* phase 1 ... */
      this.sor.barrier.wait();
      /* phase 2 ... */
      this.sor.barrier.wait();
}}}
```

Fig. 3. Simplified version of the `sor` benchmark using threads and barriers.

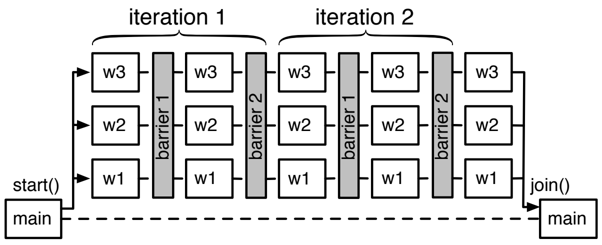

Fig. 4. One main and three worker threads while executing the example from Figure 3. Dashed lines depict periods where a thread is blocked.

The execution behavior of this program is depicted in Figure 4. The basic pattern implemented by the `Sor` class is a fork and successive join of the worker threads. For the correct functioning of the program it is important that all workers execute the loop the same number of times and that on each iteration all workers call the same barriers in the same order. If one worker would exit the loop early, for example, the barrier will block the other workers resulting in a deadlock.

The implicit dependency on the number of loop iterations and the number of `barrier.wait()` calls each worker executes makes this example hard to analyze statically. The problem is that it is non-trivial to reason about which parts of the program may be executed in parallel and which parts are ordered. For example, in the presence of aliasing it can be difficult to reason that all threads always synchronize on the same barrier object.

B. Example with Explicit Scheduling Constraints

Figure 5 shows the `Sor` class, rewritten to use explicit scheduling constraints instead of threads and barriers. The

```
 1 class Sor {
 2   Worker[] workers = new Worker[NumWorkers];
 3   int NumRounds = 100;
 4   task Begin() {
 5     for(int i=0; i<NumWorkers; i++) {
 6       this.workers[i] = new Worker(this, i);
 7     }
 8     Task end = schedule this.End();
 9     Task round = schedule this.Round(0, end);
10     round→end;
11   }
12   task End() { }
13   task Round(int count, Task later) {
14     if(count < NumRounds) {
15       Task nextRound =
16               schedule this.Round(count++, later);
17       nextRound→later;
18
19       Task barrier = schedule this.Barrier();
20       for(int i=0; i<workers.length; i++) {
21         Task phase1 = schedule workers[i].Phase1();
22         Task phase2 = schedule workers[i].Phase2();
23         phase1→barrier;
24         barrier→phase2;
25         phase2→nextRound;
26   }}}
27   task Barrier() { }
28 }
```

Fig. 5. The `sor` benchark from Figure 3 with explicit scheduling constraints.

fork/join pattern of the original code was transformed into a scheduling of the `End()` task on line 8 and passing this task object from iteration to iteration on lines 9 and 16 until the computation has finished.

The biggest change to the original example in Figure 3 is that the loop moved out of the workers and was transformed into the recursive task `Round()` in the `Sor` class. Instead of letting every worker execute the loop separately, the `Sor` object works as an orchestrator, scheduling the execution of the two work phases for each worker in each iteration.

The task `Round()` implements one iteration. After testing on line 14 whether to continue, line 16 schedules the next iteration and orders it before the original `End()` task on line 17.

Line 19 schedules the `Barrier()` task. `Barrier()` is only used for ordering purposes and has no functional behavior. On lines 21 to 25 the `Sor` class then schedules the two phases for each worker for this iteration and orders the `Phase1()` before the barrier and `Phase2()` after the barrier but before the next iteration. After the `Round()` task for one iteration has finished, the scheduler can continue to start executing all the `Phase1()` tasks of the workers.

For the `Round()` method, the analysis presented in this paper statically extracts the schedule shown in Figure 6. Task objects that are created inside `Round()` or that are passed as parameters to `Round()` are drawn as filled circles. The unfilled circle stands for the logical **`now`** task that executes `Round()`. The double-headed arrows represent the happens-before edges that are implicitly added between the creating task **`now`** and the created tasks. The other edges indicate explicit happens-before relationships resulting from the →-statements in `Round()`. Because the `Phase1()` and `Phase2()` tasks may be scheduled multiple times inside the loop, the corresponding nodes in the

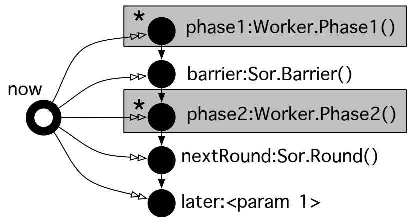

Fig. 6. Schedule statically extracted from task `Round()` from Figure 5. Double-headed arrows indicate implicit creation edges, gray boxes indicate schedule sites inside loops.

graph are marked as "multiple" as indicated by the asterisk and the gray boxes.

Given this extracted schedule, the analysis can conclude that the two phases, the barrier, the next iteration, and the `later` parameter are all ordered and therefore will never execute in parallel. The analysis can further deduce that there may be more than one `Phase1()` tasks that are not ordered with one another and that similarly, `Phase2()` may be scheduled multiple times without any internal ordering.

IV. Schedule Analysis

The goal of the schedule analysis is to determine whether at runtime two tasks may be executed in parallel or whether they are always ordered by happens-before relationships. Schedule analysis thus computes the function $Task \times Task \rightarrow Relation$ where $Relation$ is one of the following:

- *Ordered:* The two tasks are ordered if either all of their possible executions are ordered by happens-before relationships or if they can never co-exist in a single run of the program (e.g., they are scheduled in different branches of a conditional statement).
- *Parallel:* If two tasks are not ordered, they are considered (potentially) parallel.

The key insight behind our analysis is to not compute what tasks are ordered with each other but the opposite: what tasks *may be unordered*. Consider as an example the following tasks $T1()$ and $T2()$:

```
task T1() {
  Task a = schedule this.A();
  Task b = schedule this.B();
  a→b;
}
task T2() {
  Task a = schedule this.A();
  Task b = schedule this.B();
}
```

When looking for ordered-ness, discovering the statement `a→b` in task $T1()$ would make us record that tasks $A()$ and $B()$ are ordered. However, inside task $T2()$ another $A()$ task is scheduled which is unordered with respect to at least one of the $B()$'s. This discovery in $T2()$ would require us to remove the fact $A()$ `ordered-with` $B()$ from the set of known facts.

The property of "ordered-ness" is non-monotonic in the sense that finding more information may add facts as well as remove facts from the set of ordered tasks. Taking "unordered-ness" as a property, however, we can re-gain the monotonicity. If we deduce in $T2()$ that the tasks $A()$ and $B()$ may be parallel, no schedule statement and no happens-before relationship that we discover later will invalidate this fact.

The result of the schedule analysis is computed in three steps. In the first step, we compute for each task method the pairs of local task variables that are unordered with respect to each other. The information about whether two task variables are unordered is extracted from the explicit happens-before relationships present in the program code.

The second step computes for each local variable of a task method $T()$ the set of tasks that are unordered with that variable. This step is an inter-task analysis that works not only on $T()$ but also takes results computed about subtasks of $T()$ into account. The special case where a task variable is passed as a parameter to a subtask is also handled in this step.

The third and final step is to extract the pairs of unordered tasks by combining the information about unordered variable/task pairs with the tasks that may be referenced by the variables.

Schedule analysis has exactly one dependence on a points-to analysis: to resolve task method implementations at virtual call sites. If a points-to analysis is not available or considered too expensive, a quicker type-based approximation of the call-graph can be used to compute the set of target tasks at each schedule site. Since none of the benchmarks we present in this paper requires virtual dispatch for task methods, the results of the schedule analysis would be the same if we used a type-based call-graph instead of the points-to analysis.

The remainder of this section describe all three steps in more detail and discusses our approach to handling aliasing.

A. Terminology

For the analysis described in this paper, we consider *tasks* to be the same as their corresponding *task methods*. An implementation, however, may choose to add additional context sensitivity to distinguish different task objects and increase its precision.

Task variables are program variables that point to task objects. We consider a task variable to be *local* to a task $T()$ if it either comes from a schedule site inside $T()$ or if is a formal parameter of $T()$. Conditional and non-local task variables cannot be used by the analysis and must be approximated with their worst case effects as described in Section IV-E.

Because the analysis works on a static single assignment (SSA) representation of the input program, each task variable is assigned only once. Therefore, a non-parameter task variable is equivalent to the *schedule site* where it is defined. Depending on the context, we thus use the term *schedule site* interchangeably with the corresponding *task variable*.

B. Computing Unordered Variable Pairs

The first step of analyzing a task method is to extract information about the ordering of local tasks variables. For each pair of task variables a and b, this step decides whether at runtime the task objects referenced by a and b will get ordered or not.

The ordering of variables is extracted from the explicit happens-before relationships in the program code. It is important, however, to decide for each $\rightarrow$-statement whether it orders the tasks conditionally or unconditionally. As an example, consider this simplified version of the `sor` example from Figure 5 with only one barrier and only `phase2`:

```
task Round() {
  Task barrier = schedule this.Barrier();
  Task phase2 = schedule this.phase2();
  Task nextRound = schedule this.Round();
  barrier→phase2;
  phase2→nextRound;
}
```

Because all three schedule sites `barrier`, `nextRound`, and `phase2` execute unconditionally, we can conclude that the two happens-before edges have the transitive effect of ordering `barrier` and `nextRound`. This conclusion is not true, however, if any node on a transitive path may be scheduled conditionally, as in the following example:

```
task Round() {
  Task barrier = schedule this.Barrier();
  Task nextRound = schedule this.nextRound();
  while(random()) {
    Task phase2 = schedule this.Phase2();
    barrier→phase2;
    phase2→nextRound;
  }
}
```

Here, if `random()` is **`false`** the first time, the schedules site at `phase2` is never executed and the two happens-before edges are not created. Therefore, it is not guaranteed that `barrier` and `nextRound` are transitively ordered in all cases, and so they must be considered potentially parallel. However, no matter how many `Phase2()` tasks are scheduled inside the loop, we can conclude that all `Phase2()` tasks are ordered with respect to `barrier` and `nextRound` and that the `Phase2()` tasks are not ordered among themselves.

If we replace the **`while`** loop with a **`do`**/**`while`** loop, we know that the loop will be executed at least once and so we can once again conclude that tasks `barrier` and `nextRound` are ordered transitively.

In the original example, the `workers` array is initialized with a size > 0 and therefore it is known that the **`for`** loop is executed more than once. If the size of the array was unknown at compile time, the analysis would in fact conclude that `barrier` may be parallel to `nextRound` and `later`. It would still know, however, that `phase1` is ordered before `phase2`.

In [5], the authors define *genuine edges* as those edges that the analysis can rely on and they present an algorithm for how to compute them. In this paper, we assume a relation $a \leftrightarrow^* b$ stating that the task variable a is (transitively) ordered with the task variable b. For a schedule site a inside a loop that is ordered across loop iterations, this relation contains the fact: $a \leftrightarrow^* a$.

In addition to transitive ordering, we use a relation $a \neq b$. The $\neq$-relation contains two schedule sites a and b if they are different sites in the task method $T()$. Further, if a schedule site c is located inside a loop in $T()$, we say that $c \neq c\ \texttt{in}\ T()$ to capture the fact that schedule site c may be executed multiple times across different loop iterations.

UNORDEREDTASKS–INTRA

$$\frac{a \neq b\ \texttt{in}\ T() \qquad \texttt{not}\ a \leftrightarrow^* b\ \texttt{in}\ T() \qquad b\ \texttt{may schedule}\ B()\ \texttt{in}\ T()}{\textit{taskNotOrdered}(a,\ B())\ \texttt{in}\ T()}$$

UNORDEREDSUBTASKS–NOPARAM

$$\frac{a \neq b\ \texttt{in}\ T() \qquad b\ \texttt{is no param to}\ a\ \texttt{in}\ T() \qquad a \rightarrow_* b \qquad a\ \texttt{may schedule}\ A()\ \texttt{in}\ T \qquad C()\ \texttt{may be subtask of}\ A()}{\textit{taskNotOrdered}(b,\ C())\ \texttt{in}\ T()}$$

UNORDEREDSUBTASKS–WITHPARAM

$$\frac{a \neq b\ \texttt{in}\ T() \qquad a \rightarrow_* b \qquad a\ \texttt{may schedule}\ A()\ \texttt{in}\ T() \qquad c\ \texttt{is formal param}\ n\ \texttt{of}\ A() \qquad b\ \texttt{is actual param}\ n\ \texttt{to}\ a\ \texttt{in}\ T() \qquad \textit{taskNotOrdered}(c,\ D())\ \texttt{in}\ A()}{\textit{taskNotOrdered}(b,\ D())\ \texttt{in}\ T()}$$

Fig. 7. Rules for computing unordered variable/task pairs

C. *Computing Unordered Variable/Task Pairs*

In a second step, the analysis computes for every task variable the set of tasks that may be unordered with the task this variable represents. Figure 7 shows the rules for this computation.

The first rule UNORDEREDTASKS-INTRA selects for each task variable a in a task method $T()$ all unordered task variables b in $T()$. It then concludes that all tasks $B()$ that may be scheduled by b are unordered with the task variable a.

The second two rules are inter-task computations that work across task method boundaries. This is necessary because to find all tasks that may be parallel with the task represented by a task variable a we must take all possible subtasks into account.

For a given pair of schedule sites a and b, only one of the two subtask rules applies. If b is not passed as a parameter to a, the second rule UNORDEREDSUBTASKS-NOPARAM applies. The rule finds all tasks $C()$ that may be subtasks of any task $A()$ scheduled at a and concludes that those $C()$ tasks are unordered with b. The relation $C()\ \texttt{may be subtask of}\ A()$ is computed by a simple reachability analysis that starting in task method $A()$ follows all tasks that may be scheduled in all schedule sites until a fixed point is reached.

The third rule UNORDEREDSUBTASKS-WITHPARAM specializes the case where b is passed as a parameter to a in order to reduce the set of tasks that are considered potentially parallel to b. [4] Instead of simply taking all possible subtasks of $A()$, this rules selects only those subtasks of $A()$ that are unordered with respect to the corresponding parameter variable.

Applying these rules to the example schedule from Figure 6 results in the following conclusions:

[4]For reasons of brevity, this rule does not take the direction of happens-before relations into account. Our implementation splits this rule further to specialize for cases where variables are known to be ordered before or after a parameter.

$$\frac{\begin{array}{c}\text{ParallelTasks–ScheduleSite}\\ a\ \texttt{may schedule}\ A()\ \texttt{in}\ T()\\ \mathit{taskNotOrdered}(a,\ B())\ \texttt{in}\ T()\end{array}}{\mathit{parallel}(A(),\ B())}$$

$$\frac{\begin{array}{c}\text{ParallelTasks–Siblings}\\ a \neq b\ \texttt{in}\ T()\\ a\ \texttt{may schedule}\ A()\ \texttt{in}\ T()\\ b\ \texttt{may schedule}\ B()\ \texttt{in}\ T()\\ C()\ \texttt{may be subtask of}\ A()\\ D()\ \texttt{may be subtask of}\ B()\end{array}}{\mathit{parallel}(C(),\ D())}$$

Fig. 8. Computing parallel tasks

- Variable `phase1` is located inside a loop and therefore $phase1 \neq phase1$ and *not* $phase1 \leftrightarrow_* phase1$. Therefore, rule 1 can be applied and concludes that $taskNotOrdered(phase1, Phase1())$.
- Similarly, for `phase2` we can conclude that $taskNotOrdered(phase2, Phase2())$.
- All other pairs of task variables are ordered and therefore rule 1 cannot be applied to any of them.
- `phase1` is ordered before all other task variables. Therefore, `phase1` is ordered with respect to all their subtasks through the implicit creation edges. Neither rule 2 nor rule 3 applies.
- The `barrier` variable is ordered after `phase1`. Because it is not used as a parameter to `phase1`, the second rule applies and deduces that `barrier` is unordered with any subtask that may be created by `Phase1()`.
- Similar reasoning applies to the variable `phase2`, classifying it as potentially parallel with subtasks of `Phase1()` and `Barrier()`.
- The parameter `later` is passed along as a parameter to `nextRound` and ordered after `nextRound` and therefore rule 3 applies. Assuming that in this example the tasks `Barrier()`, `Phase1()`, and `Phase2()` do not schedule any subtasks, rule 3 concludes that parameter `later` is ordered with all other tasks. It is able to do this because `later` is used as an actual parameter to `nextRound` and it can map the orderings with respect to the formal parameter of `Round()` back to the schedule site and the `later` variable. This recursion is a fixed point computation that stops once the set of potentially parallel tasks stops changing.

D. Computing Unordered Task Pairs

The final step is to compute what tasks may be unordered and therefore potentially parallel to each other. Figure 8 shows the two rules that are involved in this computation.

The first rule ParallelTasks-ScheduleSite states that if a schedule site a may schedule task $A()$, task $A()$ may be parallel to all tasks $B()$ that are not ordered with the variable a. This information has been computed in the previous step.

The second rule ParallelTasks-Siblings relates all subtasks of the tasks scheduled in the same task $T()$. Without parameter passing, even if the parent tasks are ordered there is no way to order their subtasks with respect to each other. For cases with parameter passing, however, this rule is overly conservative and introduces some imprecision. In the implementation, this rule is specialized to the cases when tasks that are ordered before a parameter can be considered transitively ordered with the children of the parameter.

For the example from Figure 5, the first rule concludes that in task method `Round()` `Phase1()` is parallel to itself as is `Phase2()`. Because the only task that creates subtasks is `Round()`, the second rule does not add any additional tasks to the $parallel$ relation. The imprecision of the second rule mentioned above would consider subtasks of `End()` to be parallel with all subtasks of `Round()`. However, since `End()` does not schedule any other tasks, this imprecision does not affect the results for this example.

E. Tracking Aliasing

Task objects are normal objects that can be passed to and be returned from methods as well as stored in fields. Therefore, a program is allowed to add happens-before relationships to and from tasks that have been loaded from fields. The only two restrictions in the original model presented in [5] are that new happens-before edges must not create cycles and that a task $T()$ adding a happens-before edge between tasks $A()$ and $B()$ must be scheduled before $B()$, that is $T() \rightarrow {}^*B()$. Both restrictions together result in a well-formed schedule and guarantee that the scheduler can always choose a task for execution.

To avoid complex must-alias analysis, however, our analysis focusses on analyzing the common case of happens-before edges that are created between local task variables. Happens-before edges containing task objects that were loaded from fields are ignored by the analysis, thus over-approximating the parallelism in the program. Over-approximating parallelism is generally the safe and conservative assumption. As an example, take the effects of data races. Two tasks are allowed to write to the same data if and only if they are sequentially ordered. If the sequential execution cannot be guaranteed we must assume that both tasks are potentially executed in parallel and report a data race if they access the same data.

None of the benchmarks evaluated in Section V required task objects to be stored in fields and could therefore be analyzed without restrictions.

V. Evaluation

We evaluate the performance and effectiveness of the schedule analysis for the sequentially consistent Java compiler presented in Section II. We use several multi-threaded benchmark programs taken from three different benchmark suites to measure the cost and precision of the schedule analysis and compare the runtime overhead of an unoptimized sequentially consistent version and the optimized versions.

A. Setup of the Experiment

All experiments were run on a machine equipped with a Intel Core 2 Duo 2.8GHz and 4Gb of RAM. The compiler implementation is single threaded, however, and therefore only one core is used during the compilation.

The first six benchmarks are taken from [6]. `sor` (successive over-relaxation over a 2D grid), and `tsp` (traveling salesman problem) are data- and task-parallel applications with data access patterns of scientific codes; threads are synchronized in a fork/join style based on barriers instead of locks. `hedc` is a warehouse for scientific astrophysics data that implements a meta crawler for searching multiple Internet archives in parallel. The individual queries are handled by reusable worker threads. The programs `mol(dyn)`, `ray(tracer)`, `monte(carlo)` are multi-threaded numeric applications taken from the Java Grande benchmarks [7].

The last five benchmarks are part of the Lonestar 2.0 benchmark suite, a collection of widely-used real-world sequential applications that exhibit irregular behavior [8]. The benchmarks are parallelized using the Galois runtime system [9]. The Galois runtime provides a framework for parallelizing irregular algorithms that are organized around object-oriented pointer-based data structures such as graphs and trees..

Barnes-Hut (`barnes`) simulates the gravitational forces acting on a galactic cluster. The Boruvka's algorithm (`boruv`) computes a minimal spanning tree of an edge-weighted undirected graph. `clust` is an implementation of a well-known data-mining algorithm called Agglomerative Clustering. Delaunay triangulation `d-tri` and Delaunay mesh refinement `d-ref` compute triangulations of sets of points such that each triangle satisfies certain quality constraints.

For most cases, adapting the benchmarks to the task model was straight forward and almost purely syntactical. Only the `tsp` benchmark required some more refactoring of the original algorithm because the original code was not written in a way that fits the task model very well.

B. Benchmark Characteristics and Analysis Performance

Table I gives an overview of the size of the individual benchmarks as well as the performance of the schedule analysis. The reported numbers of application classes for the Lonestar benchmarks include the classes of the Galois runtime system.

For most cases, the schedule analysis takes only a small percentage of the overall compilation time, between 1% and 11%, with the `moldyn` benchmark being a notable exception.

For extracting the schedule from a given method, the schedule analysis uses a data-flow approach that is sensitive to the number and nesting of back-edges in the control-flow graph. `moldyn` is implemented in a way such that almost all task objects are scheduled inside a single method. This method contains multiple nested loops, in each of which task objects are scheduled and happens-before relationships are created, which causes the data-flow computation to converge only slowly. Therefore, about 98% of the time spent during schedule analysis is used for extracting the information from the bytecode and only about 2% is spent for actually analyzing the schedule. This behavior seems to be a pathological case that is related to the way this particular benchmark has been implemented; however, it also shows that further investigation in a more performant schedule extraction with better worst-case behavior is required.

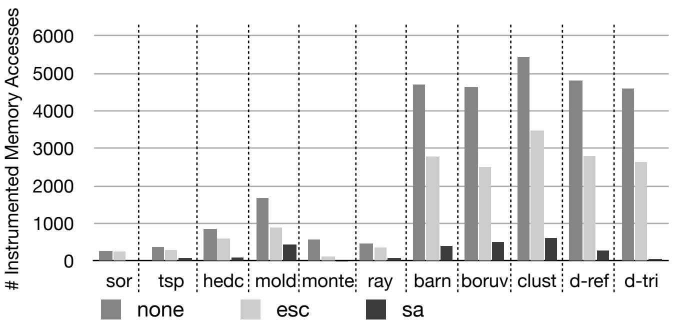

Fig. 9. Number of instrumented field and array access bytecodes.

C. Precision of the Analysis

For evaluating the effects of each individual analysis on the overall result, we have compiled the benchmarks in three different configurations:

- In the `none` configuration, the compiler has no advanced analysis information and must instrument all field and array accesses in order to guarantee sequential consistency.
- The second configuration `esc` uses escape analysis information in addition to points-to information to decide whether two memory accesses executed by different tasks may require memory barriers.
- The third configuration `sa` adds scheduling information to the points-to analysis and escape analysis to distinguish between accesses that are ordered and accesses that may happen in parallel.

Figure 9 reports the precision of the different configurations in terms of the number of memory accesses that were instrumented.

For all benchmarks, the schedule analysis significantly reduces the number of instrumentations. A large number of the instrumentations that are removed are located in single-threaded code that is executed during setup and tear-down of the applications and therefore have only relatively little effect on the overall runtime. However, in many cases the schedule analysis is able to identify and remove instrumentations in hot paths of the programs which can have a significant impact on the runtime overhead as presented in the next section.

In all benchmarks, the initial data structures are set up during the start phase and then passed to the subtasks. The subtasks perform their computation before a single-threaded teardown phase verifies the results and reports to the user. The escape analysis flags all objects that are passed to and from the parallel tasks as 'escaping'. Therefore, the compiler configuration that only uses escape information falsely identifies many memory accesses as conflicting. With scheduling information, however, the compiler can often find that shared objects are only written during the setup phase and only read in parallel.

D. Runtime Overhead of Sequentially Consistent Java

Figure 10 shows the overhead of the benchmarks compiled with the `none`, `esc`, and `sa` configurations. The baseline of this comparison is the runtime the original version that uses the relaxed Java Memory Model.

TABLE I
COMPLEXITY AND PERFORMANCE OF THE ANALYSIS.

	sor	tsp	hedc	mold	monte	ray	barn	boruv	clust	d-ref	d-tri
benchmark size											
application classes	4	9	59	13	19	18	355	381	367	364	366
library classes	40	42	68	40	44	25	82	85	78	82	69
methods in call-graph	202	258	826	251	445	225	3,670	3,792	3,769	3,736	3,700
task methods	9	2	14	11	3	6	5	3	3	2	3
analysis performance											
compilation time	2.8s	2.1s	6.2s	120.2s	3.6s	3.5s	17.9s	16.9s	17.5s	15.5s	18.0s
schedule analysis time	0.3s	0.1s	0.5s	115.8s	0.1s	0.2s	0.8s	0.3s	0.2s	0.2s	0.3s
in %	11%	7%	7%	96%	3%	6%	4%	2%	1%	1%	2%

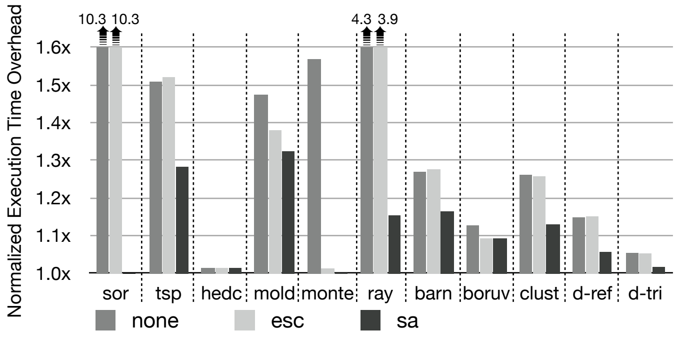

Fig. 10. Runtime overhead of unoptimized and optimized sequential consistency compared to Java's relaxed memory model.

For four out of the eleven benchmarks the fully-optimized sequentially consistent Java is within 2% of the original version whereas the configurations with no optimizations or only escape analysis show overheads between 10% and 50%. In the seven remaining cases the overhead of the fully optimized version was under 32%.

The `sor` and `ray` benchmarks show an exceptionally bad behavior for the sequentially consistent versions. The reason for this is that the authors of those benchmarks chose to never use any local variables but always access fields directly. However, instrumenting accesses inside hot loops hurt the performance especially. If the parallel tasks work on their own copies of the data the schedule analysis with the escape analysis can deduce that there are only few conflicting memory accesses and thus restore the performance of the original program.

The performance of `hedc` is mainly constrained by input/output operations which hides the overhead of sequential consistency even in the unoptimized version.

When looking at the runtime overhead it can be seen that the improvements for the Lonestar benchmarks are significantly smaller (about 5%-10% over the `none` configuration) than the improvements seen in the other benchmarks. This is, because most of the used Galois methods cannot be optimized. However, there are—often significant—sequential phases other than the setup/teardown phases that can be optimized. In `barn`, for example, each time step ends with a sequential `Advance()` task that advances all bodies by directly accessing each body's fields. The optimization removes the fences around all those field accesses which accounts for most of the improvement seen in the `barn` benchmark.

VI. Related Work

The *happens-before* ordering was first formulated by Lamport [10] and is the basis of the Java memory model [1]. Despite its significance in the memory model, in Java happens-before edges can be created only implicitly, e.g., by using **`synchronized`** blocks or **`volatile`** variables.

The ability of specifying task ordering constraints explicitly in the program code raises the question whether or not synchronization primitives such as locks are needed at all. In [11], the authors prove that it is impossible to build concurrent implementations of many algorithms and data-structures such as sets, queues, and mutual exclusion without low-level synchronization. Therefore, low-level synchronization cannot be avoided in all cases. However, algorithm designers can improve the effectiveness of the optimizing compiler by preferring explicit task-ordering over low-level synchronization wherever possible.

(Task-order Analysis) Rugina and Rinard [12] describe a pointer analysis for programs with structured fork-join style concurrency. For each program point, their algorithm computes a points-to graph that maps each pointer to a set of locations. By capturing the effects of pointer assignments for each thread, their algorithm can compute the interference information between parallel threads. Computing the interference information relies on the lexical scoping of the parallel constructs; it cannot handle unstructured parallelism.

By combining pointer and escape analysis, subsequent projects were able to extend their analyses beyond structured parallelism [13], [14]. Both analyses compute points-to information but do not directly answer as to how two tasks are executed with respect to each other. Further, the tight integration of the pointer analysis with the escape analysis and concurrency analysis is contrary to our goal of separating the concerns of schedule analysis from points-to analysis.

A *may-happen-in-parallel* (MHP) analysis can be used to determine what statements in a program may be executed in parallel [15]. Without flow sensitivity, relating two program statements is of limited use for analyzing programs with unstructured parallelism. If two threads execute the same statements but in different contexts, for example, a context insensitive MHP analysis might unnecessarily classify the statements as parallel. When the programming language is restricted to structured parallelism, as is the case for X10, an intra-procedural MHP analysis can achieve good results, however [16].

Barik [17] describes a context and flow-sensitive may-happen-before analysis that distinguishes threads by their creation site. By using threads as their model, however, they cannot exploit task ordering information available in other, non thread-based systems.

There are two general directions of research to improve the performance of sequential consistency. Systems with hardware support and pure software implementations.

(Hardware based) BulkSC [18] is a hardware implementation that enforces sequential consistency through an arbiter that determines whether groups of memory operations can be committed. Ahn et al. [19] presents BulkCompiler, a hardware-compiler interface that works with group-committing hardware to provide a whole-system high-performance sequentially consistent platform. [20] describe hardware supported conditional memory fences that decide dynamically if there is a need to stall at each fence. The conditional fence mechanism relies on compiler inserted fence instructions to achieve sequential consistency but requires an additional small hardware buffer.

(Software based) Delay-set analysis, first presented by Shasha and Snir [21], computes a minimal set of delays that guarantees sequential consistency. Those delays are enforced by inserting memory fences. By exploiting the ordering constraints of the hardware consistency model and the property of fence and synchronization operations, Lee and Padua [22] were able to optimize the number of fence instructions that the compiler must insert. In the Pensieve project, Fang et al. [23] developed various additional fence insertion and optimization algorithms. In [3] Sura et al. combine multiple analyses, namely escape, thread structure, and delay-set analysis, into a single compiler for sequentially consistent Java. Their thread structure analysis tries to identify simple cases where tasks are arranged in trivial fork/join patterns inside a single method.

The above software based implementations target programs with traditional Java threads. The unstructured nature of threads, however, makes it difficult to gather good information about the relative order in which threads execute. In contrast, the schedule analysis presented in this paper can analyze task ordering across method boundaries as well as recursive task creation. Many of the supporting analyses (especially delay-set analysis) presented in the related work, however, can be combined with schedule analysis to improve the overall analysis precision.

VII. Concluding Remarks

Optimizing compilers for parallel programs require knowledge about the scheduling of tasks at runtime to perform effective optimizations. Task-scheduling information, however, is not available in present compilers and today's systems must often assume the worst case: that all objects are shared and accessed in parallel.

This paper presents an optimizing compiler for a sequentially consistent version of Java that exploits task-ordering information gathered from the program text. Our experience shows that the schedule analysis is effective in identifying program points that potentially access shared memory in parallel.

Factoring out the schedule analysis into an independent phase has the additional benefit of making it easier to integrate additional optimizations that are not directly related to sequential consistency. This allows for previously incompatible optimizations to be applied to the same program. For example, adding a synchronization removal optimization to the compiler is trivial because it can be based on the same schedule analysis.

We believe that exposing the schedule of a program and allowing a compiler to analyze and optimize it is a necessary step towards efficient next-generation compilers for multicore systems.

References

[1] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *POPL*, 2005, pp. 378–391.

[2] J. Ševčík and D. Aspinall, "On validity of program transformations in the java memory model," in *ECOOP*, 2008.

[3] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. A. Padua, "Compiler techniques for high performance sequentially consistent java programs," in *PPoPP*, 2005, pp. 2–13.

[4] C. M. Angerer and T. R. Gross, "Static analysis of dynamic schedules and its application to optimization of parallel programs," in *LCPC*, 2010, pp. 16–30.

[5] C. M. Angerer and T.R. Gross, "now happens-before later: static schedule analysis of fine-grained parallelism with explicit happens-before relationships," in *SPLASH/Onward*, 2010, pp. 3–10.

[6] C. von Praun and T. R. Gross, "Object race detection," in *OOPSLA*, 2001, pp. 70–82.

[7] Java Grande Forum, "Multi-threaded benchmark suite," http://www.epcc.ed.ac.uk/research/java-grande/.

[8] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS'09*, 2009, pp. 65–76.

[9] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" in *PPoPP*, 2009, pp. 3–14.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, pp. 558–565, 1978.

[11] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated," in *POPL*, 2011, pp. 487–498.

[12] R. Rugina and M. C. Rinard, "Pointer analysis for structured parallel programs," in *TOPLAS*, 2003, pp. 70–116.

[13] A. Salcianu, M. C. Rinard, "Pointer and escape analysis for multithreaded programs," in *PPoPP*, 2001, pp. 12–23.

[14] M. G. Nanda and S. Ramesh, "Pointer analysis of multithreaded java programs," in *SAC*, 2003, pp. 1068–1075.

[15] G. Naumovich, G. Avrunin, and L. A. Clarke, "An efficient algorithm for computing mhp information for concurrent java programs," in *ESEC/FSE-7*, 1999, pp. 338–354.

[16] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel analysis of x10 programs," in *PPoPP*, 2007, pp. 183–193.

[17] R. Barik, "Efficient computation of may-happen-in-parallel information for concurrent java programs," in *LCPC*, 2005, pp. 152–169.

[18] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *ISCA*, 2007, pp. 278–289.

[19] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. P. Midkiff, and D. Wong, "Bulkcompiler: high-performance sequential consistency through cooperative compiler and hardware support," in *MICRO*, 2009, pp. 133–144.

[20] C. Lin, V. Nagarajan, and R. Gupta, "Efficient sequential consistency using conditional fences," in *PACT*, 2010, pp. 295–306.

[21] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst*, pp. 282–312, 1988.

[22] J. Lee and D. A. Padua, "Hiding relaxed memory consistency with a compiler," in *PACT*, 2001, pp. 824–833.

[23] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *ICS*, 2003, pp. 285–294.