Object Race Detection

Christoph von Praun and Thomas R. Gross

Laboratory for Software Technology
Department of Computer Science
ETH Zürich
8092 Zürich, Switzerland

ABSTRACT

We present an on-the-fly mechanism that detects access conflicts in executions of multi-threaded Java programs. Access conflicts are a conservative approximation of data races. The checker tracks access information at the level of objects (*object races*) rather than at the level of individual variables. This viewpoint allows the checker to exploit specific properties of object-oriented programs for optimization by restricting dynamic checks to those objects that are identified by escape analysis as potentially shared. The checker has been implemented in collaboration with an "ahead-of-time" Java compiler.

The combination of static program analysis (escape-analysis) and inline instrumentation during code generation allows us to reduce the run time overhead of detecting access conflicts. This overhead amounts to about 16-129% in time and less than 25% in space for typical benchmark applications and compares favorably to previously published on-the-fly mechanisms that incurred an overhead of about a factor of 2-80 in time and up to a factor of 2 in space.

1. INTRODUCTION

Multi-threaded execution is an attractive option for object-oriented programs that want to take advantage of multi-processor platforms. Modern object-oriented programming languages like Java include support for multi-threading directly at the language level, and there exist a number of thread packages for other languages. This move towards multi-threading is not without risks – such programs may now contain *races*, i.e., *unordered* accesses to a variable such that at least one access is a write [22]. Races can introduce ambiguity in the execution of parallel programs. The principal reason for the ambiguity is that the programmer did not introduce implicit or explicit synchronization to restrict the impact of the scheduler on the control-flow of a multi-threaded program. On certain variables, e.g. locks or volatile data, races are intentional and can be tolerated. On other variables however, races can introduce *unwanted ambiguity* and such races are termed *data races* [22] or *access anomalies* [9]. We consider data races to be *programming errors*.

There exist two principal approaches to support a programmer in recognizing and handling data races:

1. Design the programming language such that data races are by definition impossible (*prevention*).
2. Accept that the language allows data races but ensure that data races are noticed at runtime (*detection*).

Good arguments can be advanced in favor of both approaches. However, most of today's programming languages, including Java, do not adhere to either approach.

Various researchers have proposed prevention solely based on compile-time analysis and have developed extensions or annotations to allow compile-time detection of possible data races, e.g., [2, 11, 8, 30], but it will be a while before one of these improvements becomes practical and is widely adopted. In the meantime, there exist collections of "best practice" recommendations on structuring a multi-threaded program to avoid data races, but such recommendations are hardly enforceable by a compiler. Therefore another research thrust focuses on detection of races with tools that determine at runtime the presence (or absence) of data races (e.g., [26, 25, 19]).

However, the more general an environment such a runtime checker attempts to handle (e.g., by dealing with any multi-threaded program that is admitted by the operating system), the more overhead (and the less information) must be expected. We therefore restrict our attention to multi-threaded Java programs, since this language standardizes the format of multi-threading. (Another reason is that there exist portable multi-threaded programs that we can use.) This restriction also allows us to consider the complete tool chain from the compiler to the execution environment for opportunities to engage in data race detection.

Any on-the-fly approach to data race detection must balance two positions: to be practical, the run time overhead must be controlled; yet if fewer information is processed or recorded at runtime, accuracy is lost (and either real data races are overlooked or false races are reported). Compile-time approaches have almost no time constraints but may still provide only an approximation. Therefore we investigate a combination of compile-time and runtime methods: we use compile-time analysis to identify data that cannot be involved in a data race (because the data are thread-local) and then use runtime mechanisms to disambiguate the cases that static information cannot resolve. The novel aspect of this paper is the custom-tailored interaction of compiler and runtime environment. The solution proposed here has been implemented in the context of an "ahead-of-time" Java compiler, and we present empirical data that demonstrate its effectiveness.

2. ON-THE-FLY RACE DETECTION

A principal problem of known *on-the-fly* data race detection mechanisms is that they incur considerable runtime overhead in space and time. Previous work reports a slowdown by a factor of 2-80 of the original program [20, 23, 26, 25]. The high runtime overhead is one reason why on-the-fly race detectors have not enjoyed widespread use. Many previous data race detectors, however, have been overly ambitious and have not restricted the class of parallel programs that they can handle. Since our objective is to provide data race detection for Java programs, we can optimize the runtime overhead by taking advantage of the restricted context. We start with a discussion of the two major techniques for on-the-fly race detection and then illustrate how previous approaches waste cycles for many practical applications. This observation is the starting point of our integrated approach to race detection (Section 3).

2.1 Verifying access ordering

The first approach to on-the-fly data race detection is directly based on the definition of data races as unordered accesses to a variable such that at least one access is a write. The main idea then is to record information about the ordering of the current execution. A checker based on this approach must record only information that is necessary to determine possible simultaneity of accesses to variables. Essential means in this context that there must be enough information to allow the checker to distinguish an access ordering that is introduced by a scheduler decision from an ordering that is enforced by program semantics.

This detection mechanism is general and can be applied to different forms of synchronization in parallel programs, e.g., fork-join and also lock-based synchronization. The ordering of accesses must be checked relative to program synchronization, i.e., based on a model of *logical time.* However, the need to deal with logical time complicates the implementation and introduces costly data structures to track accesses and the approximative program ordering. For long-running applications, the amount of accumulated information becomes a critical issue. Various optimizations have been proposed to mitigate the memory demands [19]. Another significant problem is the execution cost per access check that verifies if the current and all previous accesses are ordered. Mellor-Crummey reduced the cost of individual access checks through a well-designed numbering of logical time [19]. Then, in later work, Mellor-Crummey employed static analysis for the Fortran source programs and thereby prevented the overhead in the first place by reducing the number of accesses that must be instrumented [20].

2.2 Verifying a locking discipline

The second principal approach to on-the-fly detection abstracts from the issues of logical time and ordering and is based on the observation that programs obeying a *locking discipline* are free from data races. A locking discipline ensures that accesses to shared data are only done inside *critical sections.* Critical sections are implemented through locks; therefore the locking discipline demands that accesses to a shared variable must be consistently protected by one common lock. A simple algorithm is able to verify this property for each access to a shared variable.

This approach, however, is restricted to parallel programs with lock-based synchronization. It has been initially proposed and used by [6, 9] and was the foundation for several implementations of on-the-fly [9, 26] as well as static race checkers [30, 8, 11]. To verify the locking discipline, one must determine the protection relation among locks and variables. A conservative approximation of this relation can be inferred from *locksets* that are associated with shared variables. A lockset keeps track of all common locks held during accesses, and this set must not become empty. For a detailed description see, e.g., Savage et al. [26]. This approach is inherently less costly than the approaches based on access ordering, as no information on program ordering must be maintained, and the code for access checks is limited to simple operations on typically small locksets. Since the space demand of approaches based on access ordering is a significant problem, we focus our attention on techniques that employ locksets.

2.3 Accuracy

The runtime overhead is not the only issue of concern for on-the-fly detection. Another concern is the loss of accuracy due to the approximative nature of the detection. Loss of accuracy is caused by data races that are possible according to the program's semantics but are not recognized by the detection algorithm, and by races that are reported yet are not possible in any program execution. However, accuracy is an inherent problem for any kind of race detection.

We use the terminology of Netzer and Miller [22] and use the term *feasible races* for all data races that are possible according to program semantics (all inputs, all schedules). An ideal race detection scheme would determine all and only feasible races at once. The precise ordering relation of accesses, as defined by the semantics of a parallel program, can however in general not be determined [21, 14]. Thus, practical race detection mechanisms must rely on an approximation and thus are (1) unable to determine all *feasible* data races and (2) also report *false* races that are artifacts of the approximation and do not represent true access conflicts.

```
class InputRace extends Thread {
   static int raceSubject;
   static int input;
   public static main(String args[]) {
      input = System.currentTimeMillis();
      new InputRace().start();
      new InputRace().start();
   }
   public void int run() {
      if (input % 2 == 0)
         raceSubject = ... ;
         // input dependent data race
   }
}
```

Figure 1: Input-dependent data race.

```
class ScheduleRace extends Thread {
   static int raceSubject;
   static int i;
   public static main(String args[]) {
      new ScheduleRace(1).start();
      new ScheduleRace(2).start();
   }
   int id;
   ScheduleRace(int id) { this.id = id; }
   public void run() {
      int tmp;
      synchronized(getClass()) {
         tmp = ++i;
      }
      if (tmp == id)
         raceSubject = ... ;
         // scheduling dependent data race
   }
}
```

Figure 2: Scheduling-dependent data race.

2.3.1 Feasible data races

An on-the-fly verification of the locking discipline is always based on a specific program execution, and thus checking is generally limited to those control flows that are allowed by the specific input. Figure 1 illustrates a program with a race that depends on the program input. It is thus desirable to have at least a race detection technique that can safely determine the feasibility of access anomalies on a certain input with a single run (i.e., has the SISE *Single Input, Single Execution* property). Dinning [9] mentions that the SISE property can be violated for programs that have *internal non-determinism* [10]. For such programs, the update sequence of variables depends on the scheduler. The actual situation is unfortunately worse such that the SISE property is not guaranteed even for programs that are *internally determinate* like the one depicted in Figure 2. The reason that the data race occurs in some runs and not in others is the existence of a *general race* that determines which data is accessed [22].

```
class Example {
   private int[] a1 = new int[100];
   private void fill(int[] a) {
      for (int i = 0; i < 100; ++i)
         a[i] = i;
   }
   void fillInstance1() {
      fill(a1);
   }
   void fillInstance2() {
      for (int i = 0; i < 100; ++i) a1[i] = i;
   }
   void fillLocal1() {
      int[] a2 = new int[100];
      fill(a2);
   }
   void fillLocal2() {
      int[] a2 = new int[100];
      for (int i = 0; i < 100; ++i) a2[i] = i;
   }
}
```

Figure 3: Race detection in different static and dynamic contexts.

2.3.2 False data races

A violation of the locking discipline might be reported although no actual race occurred. Such a *false race* occurs if access ordering is not controlled though explicit synchronization but indirectly, e.g., by passing access tokens between threads. The producer-queue-consumer setup is a good example for a programming idiom that may be subject to reporting of false races.

2.4 Opportunities for new approaches

Data race detection is frequently considered as a general, system-level mechanism, similar to low-level memory management. The race detector acts as intermediary between a black-box application and the memory system. The detector focuses solely on memory access and synchronization events. An indication of this view is that most on-the-fly race detectors (except ParaScope [19], which is based on Fortran source-code instrumentation) are based on binary instrumentation. These implementation decisions may have been motivated by efficiency concerns. But this low-level view creates unnecessary problems if access to the source code is an option for the detector. Consider the program in Figure 3 and assume that multiple threads invoke methods `fillInstance[1|2]` on the same instance of class `Example`. Since none of the methods are synchronized there could be data races involving accesses to the integer array.

If the detection system employs a low-level view, data race detection incurs more or less the same runtime overhead for all methods, as all accesses to heap-allocated data are checked (ordering or lockset). However, some of these checks are *redundant* and others are *dispensable* for practical purposes.

A check is redundant if it tests a condition that has already been evaluated, e.g., at compile-time. We observe that some of the data is thread-local and thus can be exempted from checking (stack-data, array `a2` in `fillLocal2`). Several static analyses for Java programs have been reported that conservatively determine if reference variables only refer to thread-local data [3, 4, 7, 33]. Checks for array access in `fillLocal2` are thus redundant.

A check is dispensable if it tests a condition that provides no new information. This observation is related to data encapsulation in object-oriented programs: an access to array `a1` is only possible indirectly through the invocation of one of the `fillInstance` methods. This observation can be exploited to replace 100 array access checks with one check for a method access. The checks of the array accesses inside `fillInstance2` are *dispensable* because the information "race on `fillInstance2` or `a1`" carries — for all practical purposes — the identical information as "access race on `a1` for index `0...99`".

Thus, it is promising to consider high-level language source information and the access context when deciding how and where to instrument a program for on-the-fly checking. This information provides a way to lower the overhead of on-the-fly race detection.

Another aspect that justifies a fresh look at on-the-fly race detection is that several previous race detectors were designed with scientific applications in mind. But parallel programs are also common and important for network or server-based applications. For such applications, the main targets for race detection are not arrays, but structures and objects linked through references. Java, e.g., allows a user to limit accesses to objects structures at compile-time. Such a structure then results in foreseeable access patterns at runtime. Lea [16] refers to this property as *confinement*.

2.5 Object-oriented programs

Object-oriented programs encapsulate related data, i.e., data that is read and modified together, into a common abstraction, namely the object. In a (Java-based) object-oriented world, it is thus reasonable to define objects as the unit of protection (or attention) — not individual variables. This generalization of the protection focus is safe, because all accesses to instance variables must be done through an object reference, and a conflict can be detected at the object level. From the viewpoint of data race detection, such a generalization is conservative because accesses of different threads to different instance variables cannot be distinguished. Of course, a false race will be reported in such a case, but if objects play a significant role in the data space of a program, this tradeoff may be justified. (Interestingly, since Java treats multi-dimensional arrays as arrays of objects, only accesses to 1-dimensional arrays are generalized.) We use the term *object race* in the following to refer to this conservative approximation of a data race.

3. DETECTION OF OBJECT RACES

Our data race detection mechanism is based on the verification of a locking discipline (Section 2) for objects. The locking discipline is strengthened such that all ordinary fields of an object must be consistently protected by the *same* lock, not only by *a* lock as the original discipline (Section 2.2) required. Final and volatile fields are exempted from this discipline. Section 3.1 discusses issues and optimizations that relate to static program properties. Section 3.2 addresses runtime issues. Section 4 then discusses specific implementation choices and limitations.

3.1 Confinement

Confinement is a static program property that structurally guarantees, based on data encapsulation provided by the programming language, that at most one activity/thread at a time can possibly access a given object [16, Ch. 2.3]. Thus, confinement can be exploited at compile-time to reduce the amount of instrumentation needed at runtime.

First, escape analysis is used to identify reference variables that only refer to *thread-local* objects. Subsequently, only accesses through non-thread-local reference variables are instrumented.

Second, shifting the viewpoint to the object level allows us to enhance the notion of "access" and consider, in addition to instance variables, also accesses to methods as carriers of the confinement property. Thus, as accesses to instance variables and methods through the `this` reference are transitively protected by the accessing method, we can suppress their instrumentation.

Third, transitive protection can also be assumed for objects that do not escape and are only reachable through private instance variables. Such objects are *object-local*, and races on such objects can only occur in combination with a race on the enclosing object.

Class variables are treated according to the same model as instance variables assuming protection by the respective class instance.

3.2 Ownership model

A design goal for our detection system is to carry out expensive lockset operations only for those objects that are actually shared. Thus, at runtime, we keep track of threads that have accessed an object. This information is kept in an abstract state associated with each object, i.e., the *ownership state*. State transitions are triggered by object accesses. In this section, we present the ownership model and its operational semantics as actions of the race detection checker associated with state transitions. These actions constitute the *object access protocol*. The actual implementation of this protocol is described in Section 4.2. The states and transitions of the ownership model for objects are shown in Figure 4. This model is an extension of the model of the Eraser system [26]. The model has the following states:

Virgin: Initial state after object allocation and during the execution of constructors. Concurrent access is considered an error.

Exclusive: State after constructor execution. The owner remains the same as in state *virgin*. Access by a non-owner thread is a request for *ownership transfer*. Access to objects in this state is treated as if they are thread-local, until known otherwise.

Exclusive2: State after ownership transfer; the second owner can, like the first owner, access the object as if no concurrency is present. Accesses by other threads, including the first owner, lead the object to a *shared* state.

Shared read: The object may experience concurrent read access. Accesses are tracked by updating the lockset associated with the object. No conflicts occur, even if the lockset becomes empty.

Shared modified: The object may experience concurrent read and write accesses and must thus be consistently protected by at least a single lock: the lockset associated with the object is updated with every access, and a conflict is reported if the set becomes empty.

Conflict: An access conflict has been observed for this object; it is not subject to further access checks.

The model specifically accounts for two properties that we would like to provide for Java objects. First, the distinction of the states *virgin* and *exclusive* allows us to identify conflicting accesses during object construction. Initialization semantics are particularly critical in this situation [31, 24]. Second, it is common that initialization and use of objects are logically separated (through implicit or explicit synchronization). Common programming idioms that fall into this scenario are the hand-off protocol [16] and the well-known task-queue. In such cases, the *second owner* should not be burdened with access checks necessary in a *shared* state. Thus, we defer the transition to *shared* through an intermediary state *exclusive2*. Our benchmark applications show that many objects that are exposed to concurrency are actually visited by at most two threads, and these visits occur in strict sequence (Section 5).

3.3 Accuracy

Checking races at the level of objects instead of individual variables entails the possibility of reporting false races as pointed out in Section 2.5.

An additional source of inaccuracy is introduced through the fact that the recording of access information is delayed, i.e., the creation and update of locksets is not done until a *shared* state is reached. Since we postpone noting a shared state for an object, we may introduce a dependence on the scheduler, because malicious accesses can be hidden by the *exclusive*→*exclusive2* or *exclusive2*→*shared* transitions due to unfortunate scheduler decisions. Therefore a malicious access may remain undetected. Savage et al. mention this problem as well but experience supports the view that this decision (to delay considering data as shared until proven necessary) does more good than bad for practical purposes [26].

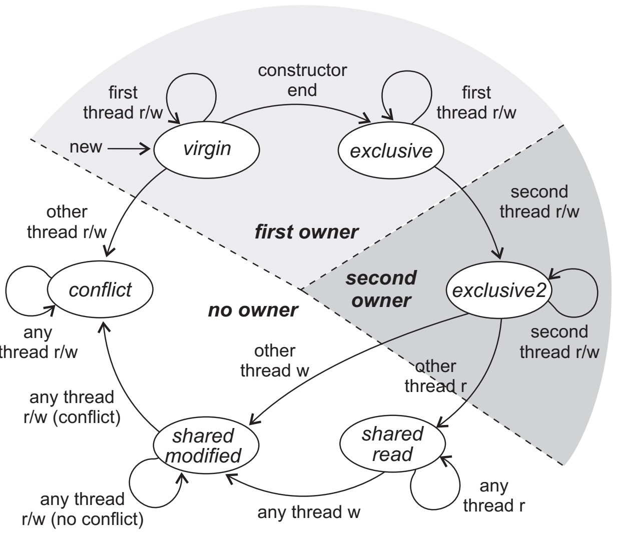

Figure 4: Ownership model.

3.4 Memory model

Generally, we would like to characterize the feasibility of data races as a program property independently of the memory model. This is however not possible, since the synchronization of program control flow and the memory view are two tangled issues, and the foundation of our race detection mechanism is based on the notion of protected program regions delimited by synchronization, i.e., critical sections. Depending on the synchronization semantics (which is part of the memory model), the scope of a critical region could deviate at runtime from its static specification in the program text. The current definition of the Java memory model allows for such enlargement of a protection region: the end of a `synchronized` block imposes a one-way memory barrier that allows, e.g., an assignment statement following this block to be hoisted inside the synchronized block [18, Ch. 8.8 and 8.13] and thus wrongly appear inside the critical region at runtime. This on-the-fly race detection mechanism thus imposes two requirements on the synchronization primitives.

- First, no actual races shall remain undetected due to the synchronization semantics. This situation could occur if a protection region is extended at compile- or runtime beyond its specification in the program code.
- Second, we would like to ensure that synchronization semantics do not mislead the race detection mechanism such that it reports additional races that were not feasible according to program semantics. For this to happen, an access would have to be moved outside its protecting region such that a violation of the locking discipline can be detected at runtime.

Our compiler and execution platform do not apply optimizations that violate these two constraints. (Although the JVM specification allows the first transformation, such a transformation is best avoided since it is likely to cause subtle problems: if the transformation has been applied, a program may work but may fail after recompilation in a different environment. The second transformation, if done by a compiler with a correct view of a program's data space, will pass by undetected. However, should the compiler's view of the data space be incorrect, our system will report a *program* error although in reality there is a *compiler* error.) A more general discussion of memory model issues in the context of data race detection can be found in [1].

Besides its influence on race detection mechanisms and accuracy, the memory model must also be considered for compile- and runtime optimizations that are enabled through the awareness of concurrency properties. An important concern is therefore the coupling of synchronization for control flow and the memory view: even if program-analysis (static) or execution-observation (dynamic) suggests that no threads act as unwanted control-flow intruders, and thus locking can be omitted, synchronization of memory view may still be necessary [4, Section 6.2].

4. IMPLEMENTATION

4.1 Overview

The implementation of an on-the-fly race detector requires additional runtime data structures that hold information, e.g., about the states and locks held by threads, the ownership state of objects, and their locksets. These data structures are themselves shared among threads and subject to data races. It is critical for an efficient implementation to make access to those meta-data thread-safe while not introducing too much overhead through additional synchronization.

Our tool chain operates as follows: Java source files are translated into JVM byte code that is analyzed and instrumented and then mapped to x86 native code. The design of the byte code reader is adopted from Bothner [5]. The instrumentation for race detection uses escape information as obtained through a data-flow analysis developed by Bogda and Hölzle [4]. The escape analysis assumes a static environment in which all classes are known at compile-time. The backend is based on LCC's pattern-matching code generator generator [12]. The Java library and parts of the runtime system including the garbage collector stem from GNU libgcj version 2.95.1 [13].

4.2 Object access protocol

The object access protocol implements the ownership model described in Section 3.2: it regulates the access to objects and arrays by different threads, and maintains meta-data structures associated with the race detection. Figure 4 defines the abstract protocol states and their transitions. There are three phases in the lifecycle of an object which have a fundamental impact on the actions and performance of object access:

First owner: Immediately prior to constructor invocation, the owner thread initializes the object header with its thread-id (first). Threads accessing this object compare their own thread-id, with the id in the object header. Equality grants immediate access, inequality activates the following access protocol: (1) If the owner thread has terminated, the accessing thread may *inherit* the object, i.e., it becomes the first owner and can immediately proceed with the execution. (2) If the owner thread is *active*, the accessing thread sends an asynchronous notification for *ownership transfer* and blocks. The owner *polls* for such notifications and eventually releases ownership to the accessing thread that now becomes the *second owner*. This idea is taken from the handling of asynchronous cache coherence messages in Shasta [27]. Polling is done at method return. (3) If the owner is *blocked* (e.g., due to a monitorenter, join, sleep, or wait), the accessing thread *steals* the object and becomes the *second owner*. Whenever an object is stolen, we make sure that no thread transits from blocked to active.

For cases (2) and (3), we require that the first owner has no active method invocations on the object. This condition is necessary because the owner loses the protection context assumed for accesses through `this` references. If there are active stack frames, (2) and (3) report an object race; then the target object is advanced to state *conflict* and has *no owner*[1].

Second owner: Like phase *first owner*, except that in cases (2) and (3), ownership is revoked altogether such that the accessed object has from then on *no owner*.

No owner: The first access in the *shared* state determines the initial lockset associated with an object. Every subsequent access verifies the locking discipline though *lockset refinement*, i.e., intersection of the locks held by the accessing thread with the lockset attached to the accessed object. A race is reported if the lockset becomes empty and there was at least one write since the object is *shared*. Subsequently, a transition to state *conflict* avoids further lockset operations.

This protocol handles access to (array) fields and methods, and our implementation conservatively treats method access like writes. Monitor access is considered independently from the ownership state of the accommodating object. From the viewpoint of race detection, lock variables are then similar to volatiles such that lock access to an object that is currently visited by another thread is not considered as an object race. For synchronized methods, first the lock, and then the object access is handled.

The ownership model has the following consequences for the implementation of the access protocol:

- The states constitute a *monotonic hierarchy*, where initial states are *strongest* with regard to the constraints they impose on object access. Strong states (*first* and *second owner*) grant privileges to the owner in terms of access performance. *Weaker* states enable access for foreign threads. For typical applications, few objects actually get into a weak state (Section 5).

[1]This transition is an implementation issue and is not shown in the conceptual model of Figure 4.

- If foreign threads access outdated ownership information (e.g., due to a weak memory model), their access behavior is strictly more conservative, i.e., they request a transition to a state that is already reached. Owner threads however, must never execute the access prologue based on outdated ownership information. In the implementation, we ensure this condition by requiring that owners themselves transit objects to weaker states. Owners that lost an object through stealing must synchronize their memory view after unblocking.
- Concurrent access can be safely recognized by the object access protocol. This fact leads to an awareness of threads on the sharing of the data they access and may provide an opportunity to explicitly establish specific coherence properties for the memory view of individual threads.

The additional synchronization introduced through the object access protocol is however not without problems:

- First, starvation is avoided through *stealing* objects; this action is a necessity of the implementation.
- Second, delaying an object access due to the access protocol must not interfere with explicit synchronization in the program such that cyclic wait conditions occur. This situation is avoided by handling all state transition requests before blocking on explicit program synchronization. If a thread is blocked, its objects may be stolen by other threads. Thus, blocked threads cannot lock out other threads in the object access protocol.
- Ownership transfer incurs an overhead of one thread-switch.

The object access protocol is not appropriate in the presence of real-time requirements because accesses to variables, methods, or arrays that are not explicitly exempted from the object access protocol (e.g., through a `volatile` modifier) may block.

4.3 Code instrumentation

The code generator inserts inline code at variable and method access sites and within method definitions to branch to routines implementing the object access protocol. The code sequences are designed such that the most frequent cases are handled most efficiently and do not require a library call. To implement this instrumentation, we enhanced the x86 code-generator and specified code templates corresponding to inline code. Instrumentation of an access to a non-this and non thread-/object-local object requires 14 instructions (3 are executed in case *exclusive*, 7 in case *conflict*). Polling of access requests at method exit costs 9 instructions (4 are executed if no message must be handled). Passing the thread-id through a register between methods requires 2 and 6 instructions for the caller and callee, respectively; this overhead could be significantly reduced if the thread-id were made available by the threading subsystem in a global register.

	exclusive	shared	conflict
class var r/w	1.4/1.2	27.5/38.4	2.0/1.6
instance var r/w	1.2/1.2	23.1/28.1	1.5/1.4
class method	1.2	4.6	1.2
inst. method stat/dyn	2.4/2.2	12.5/11.3	2.4/2.2
array (no bc) r/w	1.2/1.0	12.2/17.9	1.4/1.1
array (bc) r/w	1.2/1.2	10.0/16.8	1.3/1.3
monitor (enter+exit)	4.6	4.6	4.6

Table 1: Cost of accesses that are instrumented for race detection relative to the baseline compiler (r:read, w:write). The columns distinguish the ownership state of the accessed object.

In addition to the insertion of inline code, we allocate 4 bytes in the header of each object that are used to track ownership state and owner thread. Locksets are allocated lazily together with the monitor lock.

4.4 Micro-benchmarks

We use micro-benchmarks to determine the execution overhead of the instrumentation. The benchmark exercises different accesses inside a loop of 10^7 iterations. Execution has been done on a Pentium III/933 with 256 MB main memory. Table 1 lists the cost of access through the object access protocol. The numbers represent the relative execution time, i.e., accesses that are not instrumented for race detection correspond to 1.0. We report data dependent on the ownership state of the accessed object. For accesses to class and instance variables that are in an exclusive state, the additional cost of the state check in the object header is largely hidden by the processor cache (the header field and the accessed variable are in the same cache line). The cache effect is most evident for the performance of array accesses: the repeated race check (load and compare header of the array) is always served from the cache, whereas linear access to individual positions of a 4 MB array does not hit the cache (no data prefetch). Write accesses benefit from buffering [15]. For method accesses, most of the overhead stems from the instrumentation inside the method definition (that had otherwise an empty body in this micro-benchmark). The data for class methods include a check for class initialization which is mandatory for Java's class loading semantics. For instance methods, we distinguish invocations that are resolved statically and through dynamic dispatch; as the cost of the latter is generally higher, the relative overhead of the race check instrumentation is lower.

Access to *exclusive* and *conflict* objects is solely handled by inline codes and thus significantly faster than access to *shared* objects. The overhead for shared objects stems mainly from the implementation of locksets, which are based on the STL [28] *set* data type. The overhead of monitor access stems from maintaining a lock-nest-counter, and adding/removing the target lock from the set of locks currently held by the thread. Additional costs can occur because a thread must ensure that all pending requests for ownership transfer are processed before it possibly blocks (deadlock avoidance).

5. EXPERIENCE

For all benchmarks and instrumentation variants, we do not use optimization in our compiler to clarify the effect of the instrumentation. The application but not the Java class libraries are instrumented. This setup can be a problem for the race checker if accesses from library methods are involved in races; such races are overlooked. Our experience with the benchmark applications at hand and the detected object races (Section 5.3) demonstrate, however, that the exclusion of library procedures is not a concern for the evaluation here.

The effect of the partial instrumentation on the execution performance reported in Section 5.4 is marginal because the specific benchmarks at hand spend most of their execution in the application code. For comparison, we report also performance numbers for gcj, the GNU Java compiler [13], version 2.95.2, with different levels of optimization (O0 and O2).

5.1 Benchmarks description

The object-race checker has been applied and evaluated for five application kernels, 'elevator', 'hedc', 'mtrt', 'sor', and 'tsp'.

'elevator' is a real-time discrete event simulator. The application consists of 500 LOC and is used as an example in a course on concurrent programming. Elevators are modeled as individual threads that poll directives from a central control instance. Communication through the control board is synchronized through locks. The configuration used for this evaluation simulates 4 elevators.

HEDC is a warehouse for scientific astro-physics data developed at ETH [29]. The benchmark 'hedc' represents an application kernel that implements a meta-crawler for searching multiple Internet archives in parallel. In the benchmark configuration, 4 principal threads issue random queries to 2 archives each. The individual queries are handled by reusable worker threads. Result aggregation is followed by a short random sleep interval of 0-200 ms; this ensures that principal threads work out of sync. The application employs a library for concurrent programming by Doug Lea [17], in particular the Pooled-Executor pattern. The workload and memory access pattern of this application kernel are typical for Internet server applications and similar to applications based on alternative mechanisms such as Java Servlets.

'mtrt' is multi-thread raytrace application from the JVM98 benchmark [32]. The configuration used executed with 2 threads. Synchronization is solely applied during initialization of the threads and their copy of the "world".

'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; in 'sor', synchronization among threads is based on a barrier rather than on locks. 'sor' may thus be atypical for OO-based multi-threaded server applications that are the target of our checker; it nevertheless demonstrates the effectiveness and low overhead of our approach for applications with extensive data sharing.

5.2 Instrumentation

Table 2 lists runtime statistics for object and array accesses. Accesses are classified according to declaration properties of the accessed variable and properties of the reference variable through which the access is performed. The distinction is done to quantify the number of accesses that are instrumented.

First, access to final and volatile variables is reported separately. The remaining accesses are categorized into 'static' (class variables and methods), 'this', 'thread-local', 'object-local', and 'other'. Accesses of categories 'static' and 'this' are immediately obtained from the program. 'thread-local' and 'object-local' accesses are conservatively classified according to escape and aliasing properties of the access target. The group 'thread-local' includes targets that do not escape and do not alias with method parameters or a return value. 'object-local' includes accesses through private fields that do not escape in any method of the defining class. Escape information is computed through a context- and flow-insensitive whole program analysis according to [4]. Accesses that do not fall into any aforementioned category belong to 'other'. For array accesses of category 'other', we additionally distinguish if array elements have primitive or reference type. Every access site is reported once, namely in the first category matching from the top of the table.

Accesses that fall into category 'static', 'other', or 'other primtype' have been instrumented; the sum of these numbers is reported as 'total instrumented'.

5.3 Detection accuracy

For the evaluation of the detection accuracy, we investigate over- and under-reporting relative to actual data races that occurred during an execution. We do not account for inaccuracy due to input- and scheduling-dependences (Section 2.3.1) that may lead to races that are possible according to program semantics but not manifested in an execution history.

5.3.1 Under-reporting

Section 3.3 discusses the possibility of under-reporting, i.e., a possible scenario that lets the checker miss an object race that actually occurred at runtime. Such a situation is due to a specific schedule of accesses (first all accesses from one thread, then all accesses of the other thread) that lead to an ownership transfer instead of a race report in our model. Similarly, our model does not report object races if the other involved thread has already terminated.

Table 3 summarizes the frequence of specific incidents during object access; row 'inherit' reports the number of accesses to objects owned by threads that terminated. For the example applications, all cases of inherit resulted in benign races, such that either the access order is well-determined (e.g., access after join), or the order is irrelevant (e.g., method access to immutable objects). Repeated application runs reproduced the same race reports, although the order of race reports varied sometimes due to scheduling.

variable	reference	elevator	hedc	mtrt	sor	tsp
field accesses						
final		0.1	0.6	-	-	-
volatile		-	0.1	-	-	-
ordinary	static	-	2.2	-	-	29.8
ordinary	this	34.7	32.9	47.8	37.5	39.2
ordinary	thread-local	-	-	-	-	-
ordinary	object-local	-	-	-	-	-
ordinary	other	9.3	8.5	-	-	-
array accesses						
ordinary	thread-local	-	-	1.1	-	-
ordinary	object-local	18.5	-	7.3	-	-
ordinary	other reftype	-	0.1	3.3	31.2	5.6
ordinary	other primtype	0.1	25.9	-	31.2	24.4
method accesses						
	static	0.6	3.5	-	-	-
	this	18.5	5.4	4.2	-	1.0
	thread-local	0.2	3.1	0.9	-	-
	object-local	0.1	-	0.2	-	-
	other	18.0	17.7	35.2	-	-
total instrumented		28.0	57.8	35.2	31.2	54.2

Table 2: Runtime characteristics of object and array accesses. All numbers are reported as percentage of the overall number of accesses, or '-' if negligible or null.

The configuration of the race checker used in the tests omitted checks for accesses to arrays containing reference variables; thus races on such arrays might be overlooked. This omission of checking is not an inherent property of the mechanism, but rather an implementation decision and optimization that has been done with respect to Java's jagged array implementation.

5.3.2 *Over-reporting*

There are two major sources of over-reporting:

First, a violation of the locking discipline does not necessarily imply an object race. E.g., in a producer-consumer scenario threads are implicitly synchronized through the shared buffer ('false races', Section 2.3.2). This inaccuracy is inherent to the lockset-approach of data race detection.

Second, detecting races at the unit of objects rather than variables can lead to races reports that are not data races. Such a scenario occurs, e.g., if distinct members of the same instance are accessed by different threads. The same applies if threads access distinct regions of one array object. This factor of inaccuracy is specific to our approach of race detection.

The detection of an object-race is a runtime incident and is reported in Table 3. Category 'block' specifies the number of ownership transfers that block the accessing thread according to case (2) in the protocol (Section 4.2). 'lockset' counts the total number of lockset operations. In category 'overlapping' (cases (2) and (3)), accesses to the same object overlap in time; e.g., one thread accesses a field of an object on which another thread is executing a method. In the category 'empty lockset', an object race is concluded from a violation of the locking discipline; the conflicting accesses do not not overlap at runtime. Depending on the execution schedule, certain races can be reported in different categories in different executions. Access conflicts are

	elevator	hedc	mtrt	sor	tsp
access protocol					
block	0	78	5	2	540
lockset	18221	425	5	4501	55282206
inherit	0	60	0	1000	0
overlapping	0	44	1	2	2
empty lockset	5	63	2	4	200

Table 3: Runtime incidents of the object access protocol.

reported once per object.

For the 'elevator' application, accesses with empty lockset occur for class methods of the control panel: the methods do not keep reentrant state and thus can be independently accessed by the elevator threads; this 'object race' detected on the class instance implementing the control panel is hence benign. In addition, the invocation of the unsynchronized method `size()` in the debug output of the four elevator threads caused an object race on the `java.util.Vector` instance associated with each elevator. The application benefits from the 'second owner' concept because the elevator threads including their private auxiliary structures are initialized by a startup thread; the elevator thread accesses itself and its auxiliary structures in the role of a second owner. Suppressing the second owner concept leads to reporting 8 additional object races that are not actual races but fall into the category of object/thread initialization where access order is well-determined even without synchronization.

The majority of empty locksets reported for 'hedc' stem from accesses to stateless/immutable objects. The selection and combination of query results makes extensive use of objects for date-, time-, and number formatting; these and other factory instances are typically not altered after their initialization and thus cannot be subject of harmful races. Another common pattern that can lead to benign object races is cancellation [16]. Cancellation is used to asynchronously notify an activity (represented by the object that is subject of the race) through a method call or field update. Almost half of the overlapping object accesses are benign and are due to cancellation. The reminder resulted from access to objects that combined methods and data for different purposes: part of the object's interface is thread-safe/synchronized, another part is not. No harmful races have been possible in the actual implementation and use of this structure. Nevertheless, the race checker has called the attention to an unfavorable design that easily allowed or provoked harmful data races in the presence of inheritance and code reuse.

The raytracer 'mtrt' showed several benign object races, e.g., on a global counter of active threads (the variable is read and written by multiple threads and was apparently used for debugging; it is however not declared volatile) and another one on the output canvas object (not an actual race because the worker threads fill different regions of a pixel array managed by the canvas).

In 'sor', overlapping object access occurs when threads synchronize on a `Barrier` object that is reachable through a global variable of the class containing the `main` method. Four violations of the locking policy occur when the worker threads access the shared multidimensional `int`-arrays. These violations result from data parallelism in this applications and are not actual data races: the order of accesses to the same segments of the arrays is well-determined through barrier synchronization, not through explicit locking. The 1000 cases of 'inherit' result from a final iteration of the main thread over the data to compute a checksum.

The large number of violations of the locking policy in 'tsp' stems from different threads accessing a collection of objects representing the most promising routes found. For some cases these races are benign, i.e, they might cause unnecessary work to be done but maintain correctness of the result. Some races however involved updates that could be lost, leading to incorrect results. Thus, the checker identified a synchronization problem in the implementation of this benchmark that has led to a fix with a read-write lock.

For well-designed OO programs that respect the guidelines of data encapsulation (e.g. 'hedc'), a large fraction of false race reports could be avoided if the language allowed to explicitly declare methods as thread-safe, thus exempting them from checking. In the tests, we have explicitly suppressed the instrumentation of accesses to library methods and classes that are commonly used as if they were thread-safe. Examples are `java.io.PrintStream.println`, or access to immutable objects, e.g., of class `java.lang.String`. Checking for accesses to instance variables can be suppressed through final and volatile declarations.

5.4 Execution performance

We report on the runtime overhead due to race detection for 'mtrt', 'sor' and 'tsp'; 'elevator' and 'hedc' are not computationally bound. The runtime overhead is influenced by (1) the number of accesses that are instrumented and (2) the ownership state of the accessed object. Table 4 categorizes runtime object accesses along dimensions similar to

	elevator	hedc	mtrt	sor	tsp
not instr.					
field	34.8	33.6	47.9	37.5	39.2
array	18.5	0.1	11.6	31.2	5.6
method	19.5	16.1	5.5	-	1.0
own					
field	0.1	8.7	-	-	0.1
array	0.1	25.7	-	31.1	18.8
method	0.1	11.6	35.0	-	-
foreign					
field	9.2	0.2	-	-	-
array	-	0.1	-	-	5.6
method	8.9	0.1	-	-	-
conflict					
field	-	1.8	-	-	29.7
array	-	-	-	0.1	-
method	8.9	1.9	-	-	-

Table 4: Execution statistics of object accesses. Access frequencies are reported as percentage of the overall number of accesses, or '-' if negligible or null.

Table 1: accesses are either not instrumented ('not instr.') or instrumented. The latter category is further divided with respect to the ownership relation of accessing thread and accessed object: accesses fall in category 'own' if the accessing thread created the object or has previously gained ownership. Accesses are 'foreign' if the owner and accessing thread do not match; first time access of a thread could lead to an ownership transfer (*exclusive*→*exclusive2*), or ownership inheritance (*exclusive*→*exclusive*, *exclusive2*→*exclusive2*). If the object is *shared*, a lockset operation is necessary (Table 3, 'lockset'). Finally, accesses fall in category 'conflict' if an object race has been previously found on the access target.

The large number of non-instrumented array accesses in 'sor' stems from Java's jagged multi-dimensional array implementation: access to an array element requires one indirection per array dimension; in a two-dimensional array, the first indirection, namely an access to the reference of the array representing a row, is not instrumented.

Table 5 lists the overall number of objects ('virgin+excl.') and the ownership states reached by those. Each object in the lower categories is also included in the numbers for the upper categories. We observe that a relatively small fraction of objects is accessed by more than two threads (categories *exclusive2*, *shared*, and *conflict*).

The ownership model reflects this sharing pattern of objects in multi-threaded programs: 'mtrt' barely advances any objects beyond the *exclusive* state. The 'sor' application is different and benefits from the concept of ownership transfer: the majority of accesses go to objects in state *exclusive2*. Without the 'second owner' concept, nearly all objects (arrays that are part of the multi-dimensional array containing the application data) would transit first to state *shared*, then to *conflict*, because synchronization is based on a barrier and not on locks. Hence, the 'second owner' concept avoids a large number of false race reports in this scenario.

Table 6 shows execution times for different compilers and compiler configurations. We use 'no RD' (our compiler with-

	elevator	hedc	mtrt	sor	tsp
virgin+excl.	107	3137	6457830	1015	15053
exclusive2	37	264	9	1009	538
shared	29	127	3	6	478
conflict	5	107	3	6	202

Table 5: Allocated objects and ownership states reached.

	mtrt	sor	tsp
execution time [s]			
gcj -O0	21.2	4.1	17.6
gcj -O2	15.4	1.7	11.2
no RD	22.3	3.7	8.0
RD	41.1	4.3	18.3
relative overhead [%]	84	16	129
RD all thread-local	35.7	3.8	8.3
RD all instr.	48.5	4.8	21.2
RD single owner	41.0	4.7	16.1

Table 6: Runtime performance of compiler and instrumentation variants.

out race detection and optimizations) as base for the cost of race detection. We have executed every configuration 5 times and report the average. The data for 'RD' refer to the instrumentation and execution of race detection as explained in previous sections, including the omission of access instrumentation for 'this', thread-local, and object-local reference variables. The overhead of 'RD' is reported relative to 'no RD'.

The data for 'RD all thread-local' and 'RD all instr.' specify the execution times for various amounts of access instrumentation. For 'RD all instr.', all static accesses and accesses through reference variables are instrumented. The data for 'RD all thread-local' omit instrumentation to all heap-based objects. This number gives a lower bound for the execution performance effected by a better analysis for determining confinement of objects with respect to other objects and threads. These two artifact configurations solely serve to demonstrate the effect of instrumentation omission. Race detection is either incorrect or inefficient for such executions.

We have also executed the benchmarks in a runtime implementation that skips the 'second owner' state (Table 6, row 'RD single owner'). The 'second owner' model improves the performance of 'sor' moderatly because arrays can be accessed in state *exclusive2* instead of *conflict* (Table 1). For properly synchronized applications that are based on locks, this effect would be more pronounced if objects are, due to 'second owner', accessed in state *exclusive2* instead of *shared*. The performance of 'tsp' is degraded due to additional ownership transfers (and hence thread-switches) of objects that are subject to conflicts.

The execution times for 'tsp' fluctuated due to non-determinism in the thread-scheduling and the corresponding reduction of the cutoff boundary (for 'RD': min. 16.1, max. 20.3, avg. 18.3). The overhead (Table 6) is due to the frequent invocation of lockset operations, as is evidenced by Table 3. These invocations are the result of properly synchronized accesses to shared data.

The slowdown of 'mtrt' is, although most objects remain in 'exclusive' state (Table 5), due to the large fraction of checked method accesses (Table 2).

Our general observation for parallel applications is that the overhead of race-detection is roughly proportional to the frequency of access to actually shared objects.

The relative overhead might be higher if standard compiler optimizations are applied before instrumentation. On multiprocessor platforms, the speedup of applications may be tampered through race detection when threads accessing foreign objects are blocked.

5.5 Memory overhead

Besides the runtime effect, shifting the granularity level of race detection to objects also significantly reduces the memory overhead: four additional bytes in the object header increase the total amount of memory by at most 25%. The memory necessary for locksets is negligible for the evaluated applications: the maximum number of allocated locksets has been typically 2-50 with a peak of 423 for 'tsp'.

6. CONCLUSIONS

This paper presented a pragmatic low-cost approach to race detection for object-oriented programs. The system makes objects the unit of interest and therefore checks access conflicts at the object level. Inherent to such a strategy is the risk to report a large number of false races. Static escape analysis allows us to reduce the number of objects that must be checked. However, in practice, even more checks can be suppressed by recognizing that accesses to instance variables and methods through the `this` references do not require instrumentation. An interesting aspect is that the execution of many of the bookkeeping instructions of the checker is hidden by the organization of the platform processor.

One reason why an efficient implementation of access checks is possible is that the compiler interleaves user and runtime (checker) instructions. A pure VM could not realize these benefits; a JIT compiler that enjoys access to escape information can pursue the same strategy. The other crucial aspect is the optimization of the ownership model for objects. Programs that employ an orderly transition of objects from a first thread to a second thread pay only a low price for access checks. Of course, programs with a single user thread also benefit from this feature. Only accesses to objects that are truly shared pay the full price of maintaining ownership information. However, since the number of such objects seems to be limited for many applications, the overall runtime impact is tolerable.

Even applications that cannot accept any execution overhead can benefit from this tool during program development and testing. As our experience with a few sample program illustrates, programs sometimes exhibit races. Some races may be benign (and their reports could be suppressed by a programmer with exacter definitions of member properties), but other races may either point to a programming error or suggest an improvement in data structures or patterns. Of course, no amount of testing can replace continuous race detection.

Multi-processor PCs are today common in many environments, and developers need tools to identify problems in multi-threaded Java programs. The compiler/runtime system approach presented here is particularly well suited for programs with a structure based on objects. As object-oriented programs take increased advantage of parallelism, we expect that at some time data race detection will be as routinely performed as bound checks for array accesses.

Acknowledgments
We thank M. Corti, O. Trachsel, and D. Zogg for their contributions to the compiler system. We would like to acknowledge detailed and valuable comments from J.-D. Choi and A. Greenhouse. We also wish to thank the reviewers for their helpful comments.

7. REFERENCES

[1] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting data races on weak memory systems. In *Proc. of the Annual Int'l Symp. on Computer Architecture (ISCA'91)*, pages 234–243, May 1991.

[2] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, pages 382–400, Oct. 2000.

[3] B. Blanchet. Escape analysis for object-oriented languages - Application to Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 20–34, Denver, CO, Nov. 1999.

[4] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 35–46, Nov. 1999.

[5] P. Bothner. A gcc-based Java implementation. In *Proc. of IEEE COMPCON 97*, pages 174–178, Feb. 1997.

[6] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 21–30, Mar. 1990.

[7] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.

[8] D. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. Saxe. Extended static checking. Research Report 159, Compaq SRC, 1998.

[9] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, Santa Cruz, CA, Dec. 1991.

[10] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, Jan. 1989.

[11] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI 2000)*, pages 219–229, June 2000.

[12] C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[13] GNU Software. gcj - The GNU compiler for the Java programming language. http://gcc.gnu.org/java, 2000.

[14] D. Helmbold and C. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California, Santa Cruz, Computer Research Laboratory, Sept. 1994.

[15] Intel Corporation. Intel architecture optimization manual. http://developer.intel.com/design/PentiumIII/manuals/, 2001.

[16] D. Lea. *Concurrent Programming in Java, Second Edition.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1999.

[17] D. Lea. Package util.concurrent. http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html, 2001.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1999.

[19] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proc. of Supercomputer Debugging Workshop '91*, pages 24–33, Nov. 1991.

[20] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In B. P. Miller and C. McDowell, editors, *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 129–139, May 1993.

[21] R. Netzer and B. Miller. On the complexity of event ordering for shared-memory parallel program executions. Technical Report TR 908, Computer Sciences Department, University of Wisconsin, Madison, WI, Jan. 1990.

[22] R. Netzer and B. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[23] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, Oct. 1996.

[24] W. Pugh. Fixing the Java memory model. In *Proc. of the ACM Java Grande Conference*, pages 89–98, June 1999.

[25] M. Ronsse and K. D. Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Proc. of AADEBUG 2000, Fourth International Workshop on Automated Debugging*, pages 148–163, Aug. 2000.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP '97)*, pages 27–37, Oct. 1997.

[27] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of Seventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, pages 174–185, Oct. 1996.

[28] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett-Packard Company, 1994.

[29] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Design of a data warehouse for the HESSI solar observer. Project report, ETH Zurich, Department of Computer Science, Nov. 2000.

[30] SunSoft. lock_lint user's guide, 1994.

[31] The Java Memory Model. Mailing list and web page. http://www.cs.umd.edu/~pugh/java/memoryModel, 2000.

[32] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[33] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, Nov. 1999.