Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs

Jong-Deok Choi
IBM T. J. Watson Research Center
jdchoi@us.ibm.com

Keunwoo Lee
Univ. of Washington
klee@cs.washington.edu

Alexey Loginov
Univ. of Wisconsin - Madison
alexey@cs.wisc.edu

Robert O'Callahan
IBM T. J. Watson Research Center
roca@us.ibm.com

Vivek Sarkar
IBM T. J. Watson Research Center
vsarkar@us.ibm.com

Manu Sridharan
MIT
manu@alum.mit.edu

ABSTRACT

We present a novel approach to dynamic datarace detection for multithreaded object-oriented programs. Past techniques for on-the-fly datarace detection either sacrificed precision for performance, leading to many false positive datarace reports, or maintained precision but incurred significant overheads in the range of 3× to 30×. In contrast, our approach results in very few false positives and runtime overhead in the 13% to 42% range, making it both efficient *and* precise. This performance improvement is the result of a unique combination of complementary static and dynamic optimization techniques.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

dataraces, race conditions, debugging, parallel programs, synchronization, multithreaded programming, object-oriented programming, static-dynamic co-analysis

1. INTRODUCTION

A *datarace* occurs in a multithreaded program when two threads access the same memory location with no ordering constraints enforced between the accesses, such that at least one of the accesses is a write. In most cases, a datarace is a programming error. Furthermore, programs containing dataraces are notoriously difficult to debug because they can exhibit different functional behaviors even when executed repeatedly with the same set of inputs and the same execution order of synchronization operations. Because of the detrimental effects of dataraces on the reliability and comprehensibility of multithreaded software, it is widely recognized that tools for automatic detection of dataraces can be extremely valuable. As a result, there has been a substantial amount of past work in building tools for analysis and detection of dataraces [1, 13, 15, 17, 18, 21, 24, 27].

Most previous *dynamic* datarace detection techniques have been relatively *precise*, in that most races reported correspond to truly unsynchronized accesses to shared memory. However, these detectors incur order-of-magnitude overheads in the range of 3× to 30× [13, 18, 17, 24]. Recent approaches reduce the overhead of datarace detection, but at the cost of decreased precision. For example, monitoring dataraces at the object level rather than the memory-location level reduced overheads for datarace detection to the range of 16% to 129% [21] but resulted in many spurious race reports (see Section 9 for details).

This paper presents a novel approach to dynamic datarace detection for multithreaded object-oriented programs which is both efficient *and* precise. A key idea in our approach is the *weaker-than* relation (Section 3), which is used to identify memory accesses that are provably redundant from the viewpoint of datarace detection. Another source of reduction in overhead is that our approach does not report all access pairs that participate in dataraces, but instead guarantees that at least one access is reported for each distinct memory location involved in a datarace (see Section 2.5 for details). Our approach results in runtime overhead ranging from 13% to 42% which is well below the runtime overhead of previous approaches with comparable precision. This performance is obtained through a combination of static and dynamic optimization techniques which complement each other in reducing the overhead of our detector. Furthermore, almost all the dataraces reported by our system correspond to actual bugs, and the precise output of our tool allowed us to easily find and understand the problematic source code lines in our test programs.

Figure 1 shows the overall architecture of our approach. The first phase is an optional *static datarace analysis* which produces a *static datarace set i.e.,* a (conservative) set of statements that are identified as potentially participating in dataraces. Any statement that does not belong to the static datarace set is guaranteed to never cause a datarace during execution. If this phase is omitted, then the static datarace set defaults to all statements that contain memory accesses.

The second phase is *instrumentation*, whose goal is to insert trace statements at program points identified in the static datarace set. This insertion process can be optimized, in which case no instrumentation is inserted at *redundant trace points, i.e.,* program points

PLDI'02, June 17-19, 2002, Berlin, Germany.
Copyright 2002 ACM 1-58113-463-0/02/0006 ...$5.00.

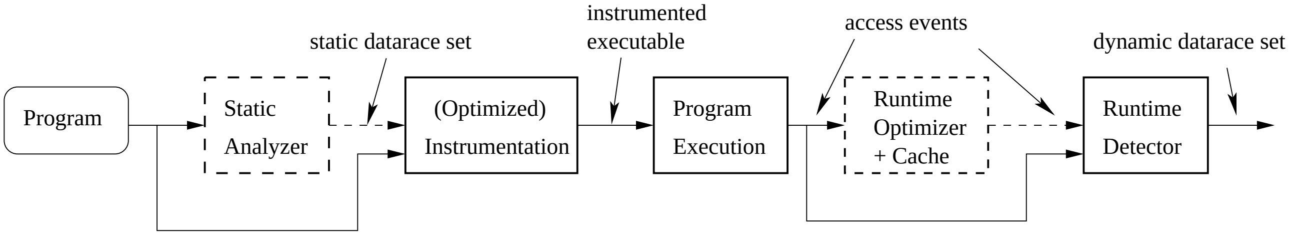

Figure 1: Architecture of Datarace Detection System

whose access events can be ignored since other (non-redundant) trace points will provide sufficient information for datarace detection. The result of the second phase is an executable that is extended with code to generate *access events* during program execution.

The third phase is an optional *runtime optimizer*, which uses a cache to identify and discard *redundant access events* that do not contain new information. Finally, the *runtime detector* examines the access events and detects dataraces during the program execution.

The instrumentation and runtime detector phases guarantee the precision of our approach, whereas the optimization phases deliver the efficiency that makes our approach practical. Our results show that it is necessary to combine all the optimization phases (static analysis, optimized instrumentation, and runtime optimizer) to obtain maximum performance. Our approach contrasts with purely *ahead-of-time* (static) datarace detection, which attempts to report dataraces that may occur in some possible program execution [1, 15, 27]. Instead, our approach detects dataraces *on-the-fly*, usually the most convenient mode for the user. If so desired, our approach could be easily modified to perform *post-mortem* datarace detection by creating a log of access events during program execution and performing the final datarace detection phase off-line.

The rest of the paper is organized as follows. Section 2 defines the conditions under which a datarace may occur and summarizes the problem statement addressed by this work. Section 3 describes the algorithm used by the runtime datarace detector. Even though it is the last phase in Figure 1, we describe the runtime datarace detector phase first because it is a mandatory phase and it provides necessary background for explaining the optimization phases. Section 4 presents the caching mechanism and key optimizations that the runtime optimizer uses to identify and delete redundant access events. Sections 5 and 6 respectively describe the static analysis and optimized instrumentation phases shown in Figure 1. In Section 7, we discuss our implementation of the ownership model [21] and its interaction with the weaker-than relation. Section 8 contains our experimental results obtained by executing a set of multithreaded Java programs on a prototype implementation of our approach. Finally, Section 9 describes related work, and Section 10 contains our conclusions.

2. DATARACE CONDITIONS AND PROBLEM STATEMENT

In this section, we first formalize the notion of dataraces, and give an example. We then formalize the problem of dynamic datarace detection and describe the set of dataraces we guarantee to detect and report.

2.1 Datarace Conditions

We define a datarace as two memory accesses which satisfy the following four conditions: (1) the two accesses are to the same memory location (*i.e.,* the same field in the same object[1]) and at least one of the accesses is a write operation[2]; (2) the two accesses are executed by different threads; (3) the two accesses are not guarded by a common synchronization object (lock); and (4) there is no execution ordering enforced between the two accesses, for example by thread `start` or `join` operations. We call these conditions the *datarace conditions*, and observe that they are different from datarace conditions assumed in past work on datarace detection for fork-join programs [1, 8]. In general, our approach is applicable to any *monitor-style* synchronization primitives supported by the programming language, operating system, or user.

2.2 Example

Figure 2 shows an example program with three threads: `main`, `T1`, and `T2`. Statements are labeled with statement numbers such as `T01`, the first labeled statement in the `main` thread. We will also use the notation `stmt:expr` to denote a field access expression within a statement. For convenience, statements that are not relevant to dataraces have been elided from this example. Note that thread `main` performs a write access on field `x.f` at statement `T01`, before creating and starting threads `T1` and `T2`.

Thread `T1` calls method `foo` which contains three accesses to object fields: a write access `T11:a.f`, a write access `T14:b.g`, and a read access `T14:b.f`. Thread `T2` calls method `bar` which contains a write access, `T21:d.f`. Let us first assume that object references `a`, `b`, `d`, and `x` all point to the same object. All the accesses to the `f` field in the example will be to the same memory location, thus every pair of them except for (`T14:b.f`, `T14:b.f`) satisfies the first of the datarace conditions.

In addition, assume that object references `T10:this`, `T13:p`, and `T20:q` all point to different objects during that execution. Then, no two statement instances belonging to different threads are guarded by the same synchronization object, satisfying the third of the datarace conditions. `T1` and `T2` are different threads without execution ordering between them via `start` or `join`, satisfying the second and the fourth of the conditions. Accesses `T11:a.f` and `T14:b.f` thus exhibit a datarace with access `T21:d.f`. Statement `T01` does not cause a datarace with the others in the example because there exists an ordering via `start` at `T04` and `T05`, not satisfying the fourth of the conditions.

Our definition of dataraces identifies both *actual* and *feasible* dataraces [20] in a given program execution. This is different from

[1] We associate only one memory location with all elements of a given array.

[2] Under certain memory models, two read accesses may also generate a datarace. The framework presented in this paper can be easily applied to such models by dropping the requirement that at least one of the accesses must be a write.

```
// THREAD MAIN
class MainThread {
      . . .
     public static void main(String args[]) {
        . . .
T01:    x.f = 100;
        . . .
T02:    Thread T1 = new ChildThread(...);
T03:    Thread T2 = new ChildThread(...);
T04:    T1.start();
T05:    T2.start();
        . . .
     }
} // class MainThread
```

```
 // CALLED BY THREAD T1
T10: synchronized void foo(...) {
T11:   a.f = 50;
T12:   . . .
T13:   synchronized(p) {
T14:     b.g = b.f;
       }
     }

 // CALLED BY THREAD T2
     void bar(...) {
T20:   synchronized(q) {
T21:     d.f = 10;
       }
     }
```

Figure 2: Example Program with Three Threads.

datarace definitions (as in [13]) that model mutual exclusion using the *happened-before relation*, and exclude feasible dataraces from their definition. For example, let us now assume that T13:p and T20:q point to the same object (which is different from the object pointed to by T10:this). Therefore the two synchronized blocks in methods foo and bar are protected by the same lock. If thread T1 acquires the lock before T2, an approach based on the happened-before relation will record the fact that statement T13 must execute before statement T20. Doing so will lead it to conclude that there is a happened-before relation from T11 to T21 (through T13), and that there is no datarace between T11:a.f and T21:d.f. In contrast, our approach reports the feasible datarace between T11:a.f and T21:d.f since it could have occurred if thread T2 acquired the lock before thread T1. In this regard, our definition of dataraces is similar to that of Eraser [24] (a more detailed comparison with the Eraser approach appears later in Sections 8.3 and 9).

2.3 Thread Start and Join Operations

As the third and the fourth of the datarace conditions indicate, there are two kinds of inter-thread serialization constructs that can be used to avoid dataraces — *mutual exclusion* (synchronized methods and blocks) and *happened-before* relations (thread start and join operations). In this section, we briefly discuss how start and join operations can be handled by a detector based on mutual exclusion, using some approximations. The rest of the paper will then present our approach to datarace detection by focusing on mutual exclusion as the sole inter-thread serialization construct.

To precisely model a join operation using mutual exclusion, we introduce a dummy synchronization object S_j for each thread T_j. The S_j locks are used solely for the purpose of datarace detection, and are not visible to the application. A dummy *mon-enter*(S_j) operation is performed at the start of T_j's execution, and a *mon-exit*(S_j) operation is performed at its end. When thread T_j's parent or any other thread performs a join operation on T_j, a dummy *mon-enter*(S_j) operation is performed in that thread after the join completes. These dummy synchronizations help the datarace detection system observe that the operations following the join cannot execute concurrently with operations in T_j.

It is difficult to model start constraints the same way, because generally one cannot know in advance how many threads will be started by each thread, or which dummy locks should be held prior to starting child threads. Instead, we use an *ownership* model to approximate the ordering constraints that arise from start operations. As in [21], we define the owner of a location to be the first thread that accesses the location. We only start recording data accesses and checking for dataraces on a location when the location is accessed by some thread other than its owner. Though approximate, this approach is sufficient to capture the ordering constraints that arise in the common case when one thread initializes some data that is later accessed by a child thread without explicit locking. (Further details on our use of the ownership model are provided in Section 7.)

2.4 Datarace Detection

We formally define *datarace detection* as follows. An *access event* e is a 5-tuple (m, t, L, a, s) where

- m is the identity of the *logical memory location* being accessed,
- t is the identity of the *thread* which performs the access,
- L is the set of locks held by t at the time of the access,
- a is the access type (one of { WRITE, READ }) and
- s is the source location of the access instruction.

Note that source location information is used only in reporting and has no bearing on our other definitions and optimizations. Given access events (or, simply, accesses) e_i and e_j, we define $IsRace(e_i, e_j)$ as follows:

$$IsRace(e_i, e_j) \iff (e_i.m = e_j.m) \wedge (e_i.t \neq e_j.t) \wedge$$
$$(e_i.L \cap e_j.L = \emptyset) \wedge (e_i.a = \text{WRITE} \vee e_j.a = \text{WRITE}).$$

A program execution generates a sequence of access events E. Performing datarace detection on this execution is equivalent to computing the value of the condition:

$$\exists e_i, e_j \in E \mid IsRace(e_i, e_j).$$

This definition does not capture the ownership model described above. Discussion of the ownership model and its effect on our design and implementation is deferred to Section 7.

2.5 Dataraces Reported

Let $FullRace = \{\langle e_i, e_j \rangle\}$ be the set of all access pairs that form a datarace during an execution. Given an execution with N accesses, any algorithm which attempts to detect all pairs in $FullRace$ must have worst-case time and space complexity $O(N^2)$ (since all possible pairs could be in $FullRace$), costs that could be prohibitive for a large sequence of accesses. To avoid these costs, our detection algorithm does not guarantee enumeration of all pairs in $FullRace$, although it still performs datarace detection as previously defined.

For each memory location m involved in a datarace, our detection algorithm reports at least one access event participating in a datarace on m. More formally, consider a partitioning of $FullRace$ by memory location into $MemRace$ sets:

$$MemRace(m_k) = \{\langle e_i, e_j\rangle \in FullRace \mid e_i.m = e_j.m = m_k\}$$

We use boolean predicate $IsRaceOn(e_i, m)$ to indicate whether the event e_i is in a pair in $MemRace(m)$:

$$IsRaceOn(e_i, m) \iff \exists e_j.\langle e_i, e_j\rangle \in MemRace(m).$$

We now define the set of dataraces reported by our approach:

Definition 1. For each m with non-empty $MemRace(m)$, our dynamic datarace detector detects and reports *at least one* access event e such that $IsRaceOn(e, m) = true$.

2.6 Debugging Support

We report a racing access e at the moment it occurs in the program, and therefore the program can be suspended and its current state examined to aid in debugging the race. Our algorithm also reports, for some previous access f with $IsRace(e, f)$, f's lockset, and often f's thread[3].

(see below). Furthermore, our static datarace analyzer discussed in Section 5 can provide a (usually small) set of source locations whose execution could potentially race with e. In our experience this information, combined with study of the source code, has been enough to identify the causes of dataraces.

To obtain full information about rarely occurring dataraces, a program record and replay tool such as DejaVu [9] can be used, where the dynamic detection runs along with DejaVu recording and the expensive reconstruction of $FullRace$ occurs during DejaVu replay. DejaVu recording incurs approximately 30% time overhead.

3. RUNTIME DATARACE DETECTION

In this section, we describe our algorithm for dynamic datarace detection. Since we do not need to report all races in a given program execution, we use two key techniques to decrease the cost of our algorithm. Our use of the *weaker-than* relation allows us to decrease the number of accesses we need to consider and save, and our representation of the access event history using *tries* [16] allows us to efficiently represent and search past accesses.

3.1 The *Weaker-Than* Relation

Given two past access events e_i and e_j, if for every future access e_k, $IsRace(e_j, e_k)$ implies $IsRace(e_i, e_k)$, e_j need not be considered when performing datarace detection on future accesses. Since e_i is more weakly protected from dataraces than e_j (or protected equally), we say that e_i is *weaker than* e_j (or e_j is *stronger than* e_i). Exploiting the weaker-than relationship between accesses allows us to greatly reduce the overhead of our datarace detection algorithm.

We now outline a sufficient condition for dynamically determining that event e_i is *weaker-than* event e_j, by using the memory location, access type, thread, and lockset information contained in each event. We add the pseudothread $t_\perp$ to the possible values of $e.t$ for a past access event e stored by our detector. $t_\perp$ means "at least two distinct threads," and we set $e_i.t$ to $t_\perp$ when we encounter some later event e_j such that $e_i.m = e_j.m$, $e_i.L = e_j.L$, and $e_i.t \neq e_j.t$. The intuition behind $t_\perp$ is that once two different threads access a memory location with the same lockset, any future access to that memory location with a non-intersecting lockset will be a datarace (unless all accesses are reads), independent of which threads previously accessed the location. Utilizing $t_\perp$ is a space optimization that simplifies our implementation, but it is also the reason why we cannot always report the specific thread for the earlier access in a datarace.

We define a partial order $\sqsubseteq$ between two threads t_i and t_j, and between two access types a_i and a_j, as follows:

$$\begin{aligned} t_i \sqsubseteq t_j &\iff t_i = t_j \vee t_i = t_\perp \\ a_i \sqsubseteq a_j &\iff a_i = a_j \vee a_i = \text{WRITE}. \end{aligned}$$

Given these orderings, we can now define the weaker-than partial order $\sqsubseteq$ for accesses:

Definition 2. For access events p and q,

$$\begin{aligned} p \sqsubseteq q \iff & p.m = q.m \wedge p.L \subseteq q.L \\ & \wedge p.t \sqsubseteq q.t \wedge p.a \sqsubseteq q.a. \end{aligned}$$

THEOREM 1 (WEAKER-THAN). *For past accesses p and q and for all future accesses r,*

$$p \sqsubseteq q \implies (IsRace(q, r) \implies IsRace(p, r)).$$

PROOF. First, $p.m = q.m$ and $q.m = r.m$ implies $p.m = r.m$. Second, $p.L \subseteq q.L$ and $q.L \cap r.L = \emptyset$ implies $p.L \cap r.L = \emptyset$. Third, $p.t \sqsubseteq q.t$ implies that $p.t = t_\perp$ or $p.t = q.t$. In either case, $p.t \neq r.t$ since $q.t \neq r.t$. (A new access r cannot have $r.t = t_\perp$.) Finally, $p.a \sqsubseteq q.a$ implies $p.a =$ WRITE or $p.a = q.a$. If $p.a = q.a \neq$ WRITE, $r.a$ must be WRITE. □

Our race detector ensures that if we detect that p is weaker than q, we at most store information about the weaker of p and q, decreasing our time and space overhead[4]. In Sections 4 and 6, we show how the weaker-than relation can also be used to filter events before they reach the detector.

3.2 Trie-Based Algorithm

In this section, we describe our runtime datarace detection algorithm and its use of *tries* to represent the event history.

3.2.1 Detection Algorithm

For each unique memory location in an access event observed by the datarace detector, we represent the history of accesses to that location using an edge-labeled trie. The edges of the trie are labeled with identifiers of lock objects, and the nodes hold thread and access type information for a (possibly empty) set of access events. The set of locks for an access is represented by the path from the root of the trie to the node corresponding to that access.

Nodes in our tries have a thread field t and an access type field a. Internal nodes which have no corresponding accesses are assigned access type READ and a special thread value $t_\top$ (meaning

[3] See Section 3.1 for an explanation of why the specific threads cannot always be reported in our approach.

[4] In the rare case that our tool reports a spurious datarace, an optimization based on the weaker-than relation could suppress the reporting of a real datarace while allowing the false positive report. Using extra locking inserted by the user to suppress the spurious report overcomes this deficiency.

"no threads"). We define the meet operator $\sqcap$ for thread information t_i and t_j and access information a_i and a_j:

$$\begin{array}{lll} \forall i. & t_i \sqcap t_i = t_i, \;\; t_i \sqcap t_\top = t_i, \;\; a_i \sqcap a_i = a_i & \\ \forall i.\forall j. & t_i \sqcap t_j = t_\bot & \text{if } t_i \neq t_j \\ \forall i.\forall j. & a_i \sqcap a_j = \text{WRITE} & \text{if } a_i \neq a_j \end{array}$$

When we encounter an access event e, we first check if there exists an access e_p in the history such that $e_p \sqsubseteq e$. This check is performed through a traversal of the trie corresponding to $e.m$, following only edges labeled with lock identifiers in $e.L$ (in depth-first order). During this traversal, we examine each encountered node's access type and thread information to see if it represents accesses weaker than e, as defined in the previous section. (The traversal procedure guarantees that the lockset and memory location weakness conditions are satisfied.) If we find such a node, then we can safely ignore e while maintaining the reporting guarantees described in Section 2. In practice the vast majority of accesses are filtered by this check.

If the weakness check fails, we check e for dataraces by performing another depth-first traversal of the trie. For each node n encountered, we have one of three cases:

Case I. The edge whose destination is n is labeled with lock identifier l_n such that $l_n \in e.L$. In this case, e shares at least one lock with all the accesses represented by n and its children. Therefore, there cannot be a datarace with any access represented by the subtree rooted at n, and we need not search any deeper in this branch of the trie.

Case II. Case I does not hold, $e.t \sqcap n.t = t_\bot$, and $e.a \sqcap n.a =$ WRITE. In this case we have a datarace, since $e.t$ differs from some previous thread which accessed $e.m$, the intersection of their lock sets is empty, and at least one access was a write. We report the race immediately and terminate the traversal.

Case III. Neither case I nor II holds, in which case we traverse all children of n.

After checking for races, we update the trie with information about e. If there is already a node n in the trie whose path to the root is labeled with the locks $e.L$, we update n with $n.t \leftarrow n.t \sqcap e.t$ and $n.a \leftarrow n.a \sqcap e.a$. If no such n exists then we add nodes and edges to create such an n, setting $n.t \leftarrow e.t$ and $n.a \leftarrow e.a$. Finally, we traverse the trie once more to remove all the stored accesses which are stronger than the newly-added access.

3.3 Implementation

We have implemented the algorithm in Java, and the code is straightforward. The algorithm runs online alongside the program. (The interface between the algorithm and the program is discussed below.)

Our implementation uses memory addresses to identify logical memory locations. Garbage collection can move objects to different addresses and reuse the same addresses for different objects. We could respond to garbage collection by augmenting the object address information stored in our data structures, but for our prototype implementation we simply use enough memory so that garbage collection does not occur.

4. RUNTIME OPTIMIZATION

The algorithm described in the previous section reads an event stream generated by the running target program. To reduce the overhead of race detection, we reduce the number of access events that need to be fed into the detector, using a combination of static and dynamic techniques. This section describes the dynamic technique: *caching* to detect redundant accesses.

4.1 Overview

The previous section describes how an access is discarded if we have already seen a "weaker" access. Experiments show that in many benchmarks almost all accesses are discarded this way. Therefore we make the check for a previous weaker access as efficient as possible, by introducing caches to record previous accesses.

There are two caches per thread, one recording read accesses and one recording write accesses. Each cache is indexed by memory location. Whenever the program performs an access to location m, we look up m in the appropriate cache. The cache design guarantees that if an entry is found, there must have been a weaker access already recorded by the algorithm, so no further work is required. If no entry is found, then we send information about the new access to the runtime detector and also add a corresponding new entry to the cache.

4.2 Cache Policy

Recall that access p is weaker than access q if $p.m = q.m \wedge p.Locks \subseteq q.Locks \wedge p.t \sqsubseteq q.t \wedge p.a \sqsubseteq q.a$. We require that if entry for access p is found in the cache when new access q is checked, then p is weaker than q.

To guarantee that $p.t \sqsubseteq q.t$, we observe that $q.t$ is simply the currently executing thread when q occurs. Therefore we use separate caches for each thread. Any p found in thread $q.t$'s cache must have $p.t = q.t$. (This also ensures that cache operations do not require synchronization.)

Because we use separate caches for reads and writes, if we find entry p when we look up the cache then certainly their access type is the same, *i.e.*, $p.a = q.a$.

To ensure that $p.Locks \subseteq q.Locks$, we monitor the set of locks currently held by each thread. Whenever the program executes `monitorexit` to release a lock l, we evict from the cache any p such that $l \in p.Locks$. This ensures that at all times, for every p in the cache, $p.Locks$ is a subset of the currently held locks. Hence when q occurs we know $p.Locks \subseteq q.Locks$ for all p in the cache.

Note that because Java synchronization blocks are reentrant, a thread might execute `monitorexit` but not actually release the lock because the lock had previously been acquired more than once. We ignore these "nested" locks and unlocks; only the last `monitorexit` on a lock object requires cache entries to be evicted.

Each cache is indexed by memory location alone. Because our policy guarantees all entries in the cache are weaker than the access being looked up, we do not actually have to check the thread ID, access type, or lock set, and they are not stored in the cache entries.

When a thread releases a lock l we need to quickly evict all the cache entries whose lock sets contain l. We exploit the nested locking discipline imposed by the Java language (although not by the bytecode language – we rely on the fact that the bytecode was generated by a Java compiler). The discipline ensures that at the time some access generated a cache entry p, if lock l was the last lock in $p.Locks$ to be acquired, then lock l will be the first of $p.Locks$ to be subsequently released ("last in, first out"). Therefore for each lock l currently held by the thread, we keep a linked list of the cache entries p where l was the last lock in $p.Locks$ to be acquired. When l is released we evict all the entries on its list from the cache. The lists are doubly-linked so that individual cache entries can be quickly removed when they are evicted due to cache conflicts.

4.3 Implementation

We use two 256-entry direct mapped caches, one for reads and one for writes, indexed by memory address. The hash function multiplies the 32-bit memory address by a constant and takes the upper 16 bits of the result. The cache code is entirely written in Java and was executed on the Jalapeño virtual machine [2]. We ensure that the Jalapeño optimizing compiler inlines all calls to the cache lookup methods in the user's program. We also used Jalapeño-specific method calls to ensure that the cache lookup code is compiled into efficient machine code (e.g., without array bounds checks). A cache lookup which results in a hit requires ten PowerPC instructions in our implementation.

5. STATIC DATARACE ANALYSIS

The static datarace analysis algorithm formulates *datarace analysis* as a conjunction of *interthread control flow analysis* and *points-to analysis* of *thread objects*, *synchronization objects*, and *access objects*. We use this formulation to compute the *static datarace set*, a set of statement pairs that *may* cause a datarace during some execution. Statements that are not part of any statement pair in the static datarace set are non-datarace statements and need not be instrumented at all. In this section, we give a brief summary of our approach to static datarace analysis. A more detailed description can be found in [11].

We first describe a *static formulation* of the datarace conditions (Section 5.1). We then describe the *interthread control flow graph (ICFG)* that we use to represent sequential and parallel interprocedural control flow (Section 5.2), and the ICFG-based *points-to analysis* that can be used to compute the static formulation of the datarace conditions (Section 5.3). Finally, we describe an extension of escape analysis [10, 4, 5, 23] that can be used to improve the precision of static datarace analysis (Section 5.4).

5.1 Datarace Conditions

For two statements x and y, the datarace conditions defined in Section 2 can be formulated conservatively as follows for static analysis[5]:

$$\begin{aligned} IsMayRace(x,y) \quad \Longleftarrow \quad & \textit{AccMayConflict}(x,y) \wedge \\ & (\neg \textit{MustSameThread}(x,y)) \wedge \\ & (\neg \textit{MustCommonSync}(x,y)) \qquad (1) \end{aligned}$$

AccMayConflict(x,y) = *true* if executions of x and y *may* access the same memory location, so we use *may* points-to information for its computation. For example in Figure 2, we use may points-to information for object references `T11:a` and `T21:d` to statically determine whether they may access the same memory location during some execution.

MustSameThread(x,y) = *true* if x and y are *always* executed by the same thread, so we use *must* points-to information on thread objects for its computation. In Figure 2, we use must points-to information on the thread objects that can run `T11` or `T21` to statically determine whether the two statements may be executed by different threads.

MustCommonSync(x,y) = *true* if x and y are *always* synchronized by at least one common lock, so we use *must* points-to information on synchronization objects for its computation. In Figure 2, we use must points-to information on the synchronization objects pointed to by `T10:this` and `T20:q` to statically determine whether the two statements may be executed under different synchronization objects.

It is worth noting that *may approximations* of *MustSameThread* and *MustCommonSync* cannot be correctly used in conservative datarace analysis, because the datarace conditions refer to the complements of these sets.

5.2 Interthread Control Flow Graph (ICFG)

The ICFG is a detailed interprocedural representation of a multithreaded program in which nodes represent instructions (*i.e.,* statements) and edges represent sequential and parallel control flow. Each method and each synchronized block has distinguished *entry* and *exit* nodes in the ICFG.

An ICFG contains four types of control flow edges: *intraprocedural* [6], *call*, *return*, and *start*. The first three types are present in a standard *interprocedural control flow graph*. Start edges are unique to the ICFG, and represent invocations of the `start()` method of a `Thread` object, which starts the thread and invokes its `run()` method. All other invocations of a `run()` method execute as part of the calling thread. (Join edges are not included in the ICFG because they are not needed for the conservative static datarace analysis performed in our approach.)

Start edges are referred to as *interthread edges*, while all other edges in the ICFG are called *intrathread edges*. The entry node that is a target of a start edge is called a *thread-root node*. An ICFG path without any interthread edges is an *intrathread path*, and an ICFG path with one or more interthread edges is an *interthread path*.

We use the *interthread call graph (ICG)* as the interprocedural abstraction of the ICFG, designed for practical and scalable analysis of large programs. An ICG node is created for each method and each synchronized block in the ICFG. The inclusion of separate ICG nodes for synchronized blocks is a notable difference between the ICG and standard call graphs.

5.3 Points-to Analysis

The points-to analysis that we employ for a static datarace analysis is a flow-insensitive, whole program analysis. In our analysis, a distinct abstract object is created for each allocation site in the program. Each abstract object represents all the concrete objects created at the same site during execution. The points-to analysis computes for each access in the program the set of abstract objects it points to along some path.

A precise must points-to analysis is expensive in general. We have devised a simple and conservative must points-to analysis based on the notion of *single-instance* statements, each of which executes at most once during an execution. An object created at a single-instance statement is called a *single-instance* object. If an access points to only one abstract object and that abstract object is a single-instance object, then the relation between the access and the object is a must points-to relation[7]. More details on the must points-to analysis are given in [11].

Let $MustPT(x))$ and $MayPT(x)$ be the *must* and *may points-to sets* of access x. We compute *AccMayConflict*(x,y) of Equation (1) as follows using points-to information:

$$\begin{aligned} & \textit{AccMayConflict}(x,y) \quad = \qquad (2) \\ & (MayPT(x) \cap MayPT(y) \neq \emptyset) \wedge (field(x) = field(y)), \end{aligned}$$

where $field(x)$ refers to the accessed field of the object (or class).

For access u, let $ThStart(u)$ be the set of thread-root nodes from whose entry nodes there exists an intrathread ICFG path to u. We compute *MustSameThread*(x,y) as follows using points-to

[5] For convenience, we ignore the fourth of the datarace conditions in Section 2, and conservatively assume that it always holds.

[6] We assume that the intraprocedural edges capture all intraprocedural control flow, including control flow arising from exceptions.

[7] We use a special "null" object to represent a null reference.

information:

$$\begin{aligned} MustThread(u) &= \bigcap_{v \in ThStart(u)} MustPT(v.\texttt{this}) \\ \textit{MustSameThread}(x,y) &= \\ &\quad (MustThread(x) \cap MustThread(y) \neq \emptyset), \end{aligned} \tag{3}$$

where $v.\texttt{this}$ denotes the `this` pointer of thread-root node v. For node $n \in ICG$, let $Synch(n) = true$ if n is a synchronized method or block, and let u_n be the access of the synchronization object if $Synch(n) = true$. Also, let $Pred(n)$ be the set of *intrathread* predecessor nodes of n on ICG. We compute $MustSync(v)$ by the following set of dataflow equations:

$$\begin{aligned} Gen(n) &= \begin{cases} MustPT(u_n) & \text{if } Synch(n) \\ \emptyset & \text{otherwise} \end{cases} \\ SO_o^n &= SO_i^n \cup Gen(n),\ SO_i^n = \bigcap_{p \in Pred(n)} SO_o^p \\ MustSync(v) &= SO_o^n, \forall v \in n. \end{aligned}$$

Now, we compute *MustCommonSync*(x, y) as follows:

$$\begin{aligned} \textit{MustCommonSync}(x,y) &= \\ &\quad (MustSync(x) \cap MustSync(y) \neq \emptyset). \end{aligned} \tag{4}$$

Finally, we compute $IsMayRace$ in Equation 1 by combining Equations 2, 3, and 4.

5.4 Extending Escape Analysis

Past work on escape analysis normally identifies objects as *thread-local* when they are never reachable from threads other than the thread that created them. A thread-local object can never participate in a datarace.

Java code frequently uses objects associated with a thread T which do not follow the above pattern but which are not susceptible to data races. In particular, we say an object O is "thread-specific" to T if all accesses to O are performed while T is being constructed (and before T starts running), or by T itself. References to such objects are typically stored in fields of the T object and hence escape to the thread creating T, and are not thread-local as described above. Because this usage is common, we extended our static analysis to identify some thread-specific objects.

We have implemented a simple, but effective, approximation algorithm to compute the thread-specific objects. First, we define the *thread-specific methods* recursively as follows: (1) `<init>` methods of thread objects, and `run` methods that are not invoked explicitly (*i.e.,* invoked only as a result of the thread being started) and (2) a non-static method all of whose direct callers themselves are thread-specific non-static methods passing their `this` references as the `this` reference of the callee.

Second, we define the *thread-specific fields* as the fields of a thread that are only accessed via `getfield`/`putfield` operations on the `this` reference of a thread-specific method. Finally, we define an *unsafe thread* as a thread whose execution may start before its initialization completes. A thread object is conservatively identified as unsafe if its constructor can transitively call `Thread.start` or if the `this` reference escapes from the constructor. (A thread is *safe* if it is *not* unsafe.)

Based on these definitions, we say an object is thread-specific to `T` *if* `T` is safe and the object is only reachable from thread-specific methods of `T` or through thread-specific fields of `T`. Accesses to a thread-specific object of a safe thread cannot be involved in a datarace. Moreover, accesses to thread-specific fields cannot be involved in a datarace.

6. COMPILE-TIME OPTIMIZATIONS

Our static datarace analysis phase improves the performance of our dynamic detector by eliminating from consideration statements that can never participate in a datarace. Another approach to compile-time optimization stems from the weaker-than relation defined in Section 3: if the execution of a statement always generates an access that will be discarded because a previous access is weaker, the statement need not be instrumented. In this section, we describe how we use a static form of the weaker-than relation and a loop peeling transformation to avoid inserting instrumentation that we can prove will only produce redundant access events.

6.1 Static Weaker-Than Relation

Let $Events(S)$ denote the set of access events generated by instrumentation statement S in a given execution. We define the static weaker-than relation for statements as follows:

Definition 3. S_i is *weaker than* S_j, written as $S_i \sqsubseteq S_j$, iff for all $e_j \in Events(S_j)$ in any given execution, there exists $e_i \in Events(S_i)$ in the same execution such that (1) $e_i \sqsubseteq e_j$, where $e_i \sqsubseteq e_j$ as defined in section 3, and (2) there exists no thread `start()` or `join()` between e_i and e_j.

A sophisticated interprocedural analysis would be required to determine $S_i \sqsubseteq S_j$ for arbitrary S_i and S_j. However, we have developed a conservative and effective analysis for computing $S_i \sqsubseteq S_j$ when S_i and S_j belong to the same method.

We model the instrumentation which generates access events using a pseudo-instruction $trace(o, f, L, a)$, where o is the object being accessed, f is the field of the object being accessed, L is the lock set held during the access, and a is the access type (READ or WRITE). All operands are treated as *uses* of their values. For accesses to static fields, o represents the class in which the field is declared, and for accesses to array elements, f represents the array index. Thread information is not explicitly modelled in the $trace$ instruction since we do not attempt to optimize across thread boundaries (thread information is available to the instrumentation code at runtime). We insert a $trace$ pseudo-instruction after every instruction which accesses a field of an object, a static field, or an array element (optionally using information from static datarace analysis to eliminate consideration of instructions which cannot be involved in dataraces).

After insertion, we attempt to eliminate $trace$ pseudo-instructions using the static weaker-than relation. First, we define $Exec(S_i, S_j)$ for statements S_i and S_j of the same method as follows:

Definition 4. $Exec(S_i, S_j)$ is true iff (1) S_i is on every intraprocedural path that contains S_j, and (2) there exists no method invocation on any intraprocedural path between S_i and S_j.

The first condition indicates that whenever S_j executes in an execution instance of the method, S_i also executes. Two well-known concepts can be used for computing $Exec(S_i, S_j)$: S_i *dominates* S_j, written $dom(S_i, S_j)$, and S_i *post-dominates* S_j, written $pdom(S_i, S_j)$. In our experiments, we used dom. (It is very difficult to prove that one statement post-dominates another in Java, because almost any statement can throw an exception, and therefore we suspect that $pdom$ would not be effective.) The second condition guarantees that no path between S_i and S_j will contain `start()` or `join()`.

With $Exec$, the static weaker-than relation can be decomposed into the following easily verifiable conditions (notation to be explained):

$$\begin{aligned} S_i \sqsubseteq S_j \quad \Longleftarrow \quad & Exec(S_i, S_j) \wedge a_i \sqsubseteq a_j \wedge outer(S_i, S_j) \\ & \wedge valnum(o_i) = valnum(o_j) \wedge f_i = f_j. \end{aligned}$$

To show that a statement $S_i = trace(o_i, f_i, L_i, a_i)$ always generates an event e_i weaker than any e_j produced by

$$S_j = trace(o_j, f_j, L_j, a_j),$$

we must show that

$$e_i.t \sqsubseteq e_j.t \wedge e_i.a \sqsubseteq e_j.a \wedge e_i.L \subseteq e_j.L \wedge e_i.m = e_j.m.$$

Intraprocedurally, $e_i.t$ will always equal $e_j.t$, and we can directly check $a_i \sqsubseteq a_j$ which implies $e_i.a \sqsubseteq e_j.a$. We check that $e_i.L \subseteq e_j.L$ using the nesting of Java's synchronization blocks. Specifically, we verify the condition $outer(S_i, S_j)$, which is true if and only if S_j is at the same nesting level in synchronization blocks as S_i *or* at a deeper level within S_i's block. Finally, to show that $e_i.m = e_j.m$, our analysis checks that

$$(valnum(o_i) = valnum(o_j)) \wedge (f_i = f_j),$$

where $valnum(o_i)$ is the *value number* of the object reference. If all of these conditions hold, then $S_i \sqsubseteq S_j$, and therefore we can safely eliminate S_j.

6.2 Implementation

In this section, we briefly describe the implementation infrastructure that we use for optimized instrumentation. The instrumentation and the analysis of the weaker-than relation is performed during the compilation of each method by the Jalapeño optimizing compiler [2]. We created a new instruction in the high-level intermediate representation (HIR) of the compiler corresponding to our $trace$ pseudo-instruction, and these instructions are inserted as previously described. After the insertion of the $trace$ statements, conversion to *static single assignment* (SSA) form is performed, during which the dominance relation is computed. Elimination of redundant $trace$ statements is then performed based on the static weaker-than relation, utilizing an existing value numbering phase. The remaining $trace$ statements are marked as having an *unknown* side effect to ensure they are not eliminated as dead code by Jalapeño's other optimization phases unless they are truly unreachable.

After the completion of some of Jalapeño's HIR optimization phases, we expand each $trace$ statement into a call to a method of our dynamic detector, and we force Jalapeño to inline this call. Jalapeñothen optimizes the HIR again. Finally, the HIR representation is converted to lower-level representations (and eventually to machine code) by the compiler, without further instrumentation-specific optimization.

6.3 Loop Peeling

Loops can be a key source of redundant access events. For example, in the loop in Figure 3 consisting of statements `S10` through `S13`, statement `S13` will produce redundant access events after the first iteration of the loop, since the information is the same as that recorded in the first iteration. However, two issues make these redundant events difficult to statically eliminate. Our redundancy elimination based on the static weaker-than relation cannot be applied to remove the instrumentation, since the information produced in the first iteration of the loop is not redundant. Furthermore, we cannot perform standard loop-invariant code motion to hoist the instrumentation outside the loop, because statement `S11` is a *potentially excepting instruction* (PEI); it may throw an exception and bypass the remaining instructions of the loop. Thus statement `S13` is not guaranteed to execute even if the loop condition is initially true. PEIs occur frequently in Java because of safety checks such as null-pointer and array bounds checks.

We reduce the generation of redundant access events in loops using a *loop peeling* program transformation. This transformation creates a new copy of the body of the loop for the first iteration and utilizes the original body for the remaining iterations. Statements `S20` through `S26` show the result of loop peeling and our existing redundancy elimination applied to the loop of `S00`. The `if` statement at `S20` is needed to guard against the possibility of the loop not executing at all. The `for` statement at `S24` is modified to ensure that the loop will not execute the first iteration, which is now executed by statements `S21` through `S23`. After the loop peeling, the $trace$ statement in the loop body can be eliminated since statement `S23` is statically weaker. The resulting code traces the write access to $a.f$ at most once, achieving the goal of eliminating the instrumentation from the loop. We are unaware of previous work that performs this type of program transformation to decrease the cost of instrumentation.

```
// Before optimization.
S00: A a;
S10: for(...) {
S11:   PEI
S12:   a.f = ...;
S13:   trace(a, f, L, W)
     }

// After optimization.
S20: if (...) {
S21:   PEI
S22:   a.f = ...;
S23:   trace(a, f, L, W);
S24:   for (...) {
S25:     PEI
S26:     a.f = ...;
       }
     }
```

Figure 3: Example of Loop Peeling Optimization

7. OWNERSHIP MODEL

All of the preceding discussion ignores the effects of the "ownership model" introduced in Section 2.3. Here we briefly consider how the ownership model interacts with our other machinery.

7.1 Implementation

We modified our runtime race detector to record for each memory location an owner thread t_o, the first thread to access the memory location. Every time the location is accessed we check to see if the current thread is t_o, and ignore the access in that case. The first time the current thread is not t_o, we say the memory location becomes *shared*; we set t_o to $\perp$ and send this access event and all subsequent events on to the rest of the detector, as described in section 3. Essentially the access event stream is filtered to only include accesses to memory locations in the shared state.

7.2 Interactions with Weaker-Than Relation

The run-time and compile-time optimization phases rely on the concept of one access event e_1 being "weaker-than" another event e_2, in which case e_2 can be suppressed. Unfortunately, in the presence of the ownership model, the definitions of $IsRace$ and *weaker-than* in section 3.1 are not sufficient to guarantee that e_1 *weaker-than* e_2 implies e_2 can be suppressed. The difficulty arises when an event e_1 is sent to the detector while $e_1.m$ is in the owned state, and then $e_1.m$ changes to the shared state before e_2 occurs. In this situation e_2 should not be suppressed.

For run-time optimization (i.e., the cache), we can avoid this problem by forcibly evicting a location m from each thread's cache when it becomes shared.

It is harder to avoid this problem in compile-time optimization. Given two statements S_1 and S_2, it is generally difficult to prove that the accessed location's state cannot change from "owned" to "shared" between S_1 and S_2. Introducing a dynamic check of the ownership state at S_1 or S_2 would eliminate the benefit of the optimization. The only truly sound compile-time approach would be to use the *post-dominance* relationship; i.e., when S_2 post-dominates S_1 and the access at S_2 is guaranteed to be weaker than S_1, remove the instrumentation at S_1. This is safe because if the object is owned at S_2, and therefore the access is suppressed, then the object must also have been owned at S_1 and that access can also be suppressed. Unfortunately, as previously noted, post-dominance between S_1 and S_2 almost never holds in Java because almost any bytecode instruction can throw an exception. (This might be less of a problem in other languages.)

Our actual approach is to simply ignore the interaction between weaker-than and the ownership model, for both static and dynamic optimizations. This means that in theory our tool may inadvertently suppress accesses and thus fail to report races. However, we did not observe any such problems in practice; in our experiments we verified that the same races were reported whether the optimizations using the "unsafe" weaker-than relation were enabled or disabled.

8. EXPERIMENTAL RESULTS

Here we present evidence supporting our two major claims: that our definition of dataraces captures truly unsynchronized accesses with fewer "false alarms" than alternative definitions, and that those dataraces can be detected with modest overhead — especially compared to other datarace detection implementations.

8.1 Program Examples

Table 1 lists the programs used in our experiments.

We derived `sor2` from the original `sor` benchmark by manually hoisting loop invariant array subscript expressions out of inner loops. This optimization could be performed by a compiler using only intraprocedural analysis, but it is not implemented in Jalapeño, and it has significant impact on the effectiveness of our optimizations. We modified `elevator` slightly to force it to terminate when the simulation finishes (normally it just hangs). [8]

The `elevator` and `hedc` benchmarks are interactive and not CPU-bound, and therefore we do not report performance results for these benchmarks.

8.2 Performance

Table 2 shows the runtime performance of our algorithm and some selected variants to demonstrate the impact of each of our optimizations. "Base" records the performance of each example without any instrumentation (and without loop peeling). "Full" is our complete algorithm with all optimizations turned on. "NoStatic" is "Full" but with the static datarace detection turned off, so all access statements are potential dataraces. "NoDominators" is "Full" with the static weaker-than check disabled; it also disables loop peeling (which is useless without that check). "NoPeeling" turns off loop peeling only. "NoCache" disables the cache.

In `mtrt` without static datarace detection, we instrument so many accesses that Jalapeño runs out of memory before the program terminates.

For each configuration, we ran the program five times in one invocation of the VM and reported the best-performing run. We enabled full optimization in Jalapeño but disabled adaptive compilation. Jalapeño was configured to use a mark-and-sweep garbage collector, but we set the heap size to 1GB of RAM so no GC actually occurred. Our test machine had a single 450MHz POWER3 CPU running AIX.

These overheads are lower than for any previously reported dynamic datarace detection algorithm. The benefits of each optimization vary across benchmarks, but each optimization is vital for some benchmark. Programs such as `tsp`, with loops involving many method calls and even recursive method calls, benefit greatly from the cache. Programs such as `sor2`, which are dominated by loops over arrays, benefit most from dominator analysis and loop peeling.

We did not measure space overhead directly; Jalapeño mixes program data with virtual machine data, making space measurements difficult. Our instrumentation consumed the most space for `tsp`, requiring approximately 16K of memory per thread (for 3 threads) and 7967 trie nodes holding history for 6562 memory locations. (We have a scheme for packing information for multiple locations into one trie which we cannot present due to space limitations.) We estimate the total amount of memory used by instrumentation for `tsp` to be about 500K.

8.3 Accuracy

Table 3 records the number of objects for which we report dataraces using our algorithm and some selected variants. (We normally output each object field on which a datarace occurs; for comparison purposes, here we count only the number of distinct objects mentioned.) "Full" is our complete, most precise algorithm. "FieldsMerged" is a variant of our algorithm where we do not distinguish different fields of the same object, so one thread accessing $o.f_1$ might appear to datarace with another thread accessing $o.f_2$ if they do not hold a common lock. (Static fields of the same class are still distinguished.) "NoOwnership" is another variant of "Full" which does not wait for a location to be touched by multiple threads before starting to monitor its accesses.

We report two dataraces in `mtrt`. Accesses to the field `RayTrace.threadCount` are not synchronized, causing its value to potentially become invalid; fortunately its value is not actually used. There are also unsynchronized accesses to `ValidityCheckOutputStream.startOfLine` in the SPEC test harness, which could result in incorrect output.

`tsp` has a serious datarace on `TspSolver.MinTourLen`, which can lead to incorrect output. We also report dataraces on fields of `TourElement`, which cannot in fact happen due to higher-level synchronization.

The dataraces we report in `sor2` are not truly unsynchronized accesses; the program uses barrier synchronization, which is not captured by our algorithm.

The dataraces we report in `hedc` are all true unsynchronized accesses and have two causes. The size of a thread pool is read and written without appropriate locking, which could cause the pool size to become invalid. More seriously, there is an unsynchronized assignment of null to field `Task.thread_`, which could cause the program to die with a `NullPointerException` if the `Task` completes just as another thread calls `Task.cancel`. This would be nearly impossible to find during normal testing and debugging. In fact, previous work [21] mistakenly classified this datarace as benign (possibly because they had to sort through a number of spurious datarace reports).

If we fail to distinguish fields, in `hedc` we produce spurious

[8] We obtained all these examples from Praun and Gross, to whom we owe great thanks.

Example	Lines of Code	Num. Dynamic Threads	Description
mtrt	3751	3	MultiThreaded Ray Tracer from SPECJVM98
tsp	706	3	Traveling Salesman Problem solver from ETH [21]
sor2	17742	3	Modified Successive Over-Relaxation benchmark from ETH [21]
elevator	523	5	A real-time discrete event simulator
hedc	29948	8	A Web-crawler application kernel developed at ETH [21], using a concurrent programming library by Doug Lea.

Table 1: Benchmark programs and their characteristics

Example	Base	Full	NoStatic	NoDominators	NoPeeling	NoCache
mtrt	9.0s	10.9s (20%)	Out of Memory	10.9s (21%)	10.9s (21%)	11.4s (26%)
tsp	10.0s	14.2s (42%)	27.5s (175%)	15.7s (57%)	15.7s (57%)	381.7s (3722%)
sor2	2.4s	2.7s (13%)	2.7s (13%)	9.8s (316%)	7.7s (226%)	3.2s (37%)

Table 2: Runtime Performance

Example	Full	FieldsMerged	NoOwnership
mtrt	2	2	12
tsp	5	20	241
sor2	4	4	1009
elevator	0	0	16
hedc	5	10	29

Table 3: Number of Objects With Dataraces Reported

race reports in the `LinkedQueue` class where some fields are immutable and accessed without synchronization and others are not. It also produces spurious warnings for `MetaSearchRequest` objects where some fields are thread-local and others are shared and require synchronization. In `tsp` we report additional spurious dataraces on fields of `TourElement`.

In all benchmarks, NoOwnership reports many spurious dataraces when data is initialized in one thread and passed into a child thread for processing.

Previous work such as Eraser [24] and object datarace detection [21] uses a looser definition of dataraces, where a datarace is deemed to have occurred on a location m if there is no single common lock held during all accesses to m. This approach produces spurious datarace reports in `mtrt`, where variables holding I/O statistics are accessed by two child threads holding a common lock `syncObject`, but also by a parent thread after it has called `join` on the two child threads but without any other synchronization. Our scheme for representing `join` introduces pseudolocks S_1 and S_2; the three threads access the variables with lock sets $\{S_1, \texttt{syncObject}\}$, $\{S_2, \texttt{syncObject}\}$ and $\{S_1, S_2\}$. We report no datarace because these locksets are mutually intersecting, although they have no single common lock.

In summary, for these benchmarks, most of the dataraces we report are true unsynchronized accesses, and most of those correspond to real bugs. Using a less strict definition induces significantly more spurious reports.

9. RELATED WORK

Past research on datarace detection can be classified as *ahead-of-time*, *on-the-fly*, or *post-mortem*. These approaches offer different trade-offs along *ease-of-use*, *precision*, *efficiency*, and *coverage* dimensions.

Ahead-of-time datarace detection is usually performed in *static datarace analysis* tools which yield high coverage by considering the space of all possible program executions and identifying dataraces that might occur in any one of them. Flanagan and Freund's datarace detection tool is a static tool for Java [15] which tracks synchronization using extended type inference and checking. Guava is a dialect of Java that statically disallows dataraces by preventing concurrent accesses to shared data [3]. Only instances of classes belonging to the *class category* called *monitor* can be shared by multiple threads. By serializing all accesses to fields or methods of the same shared data, Guava can prevent dataraces. Boyapati and Rinard propose a system of type annotations for Java that ensures a well-typed program is datarace-free and allows the programmer to write a generic class and subclass it with different protection mechanisms. [6].

Warlock is an annotation-based static datarace detection tool for ANSI C programs [27], which also supports lock-based synchronization. Aiken and Gay's work statically detects dataraces in SPMD programs [1]. Since SPMD programs employ barrier-style synchronizations, they need not track locks held at each statement. The static datarace analysis employed as part of our datarace detection is based on points-to analysis of reference variables [7, 26]. The primary advantage of a static analysis approach is its efficiency due to the fact that it incurs no runtime overhead. However, this advantage is mitigated in practice by severe limitations in precision (due to false positive reports) and ease-of-use (due to the requirement of presenting a whole program to the static analysis tool, sometimes augmented with annotations to aid the analysis).

The key advantage of dynamic analysis approaches such as on-the-fly and post-mortem datarace detection is the precision of the results (few or no false positives), but in past work this advantage usually came at a high cost in efficiency. A dynamic approach also has more limited coverage than a static approach because it only reports dataraces observed in a single dynamic execution. In some cases, dynamic tools can improve coverage by considering alternate orderings of synchronization operations that are consistent with the actual events observed in the original program execution [24].

Dinning and Schonberg introduced the idea of detecting dataraces based on a proper locking discipline [14]. Their system employed a detection approach based on both the happened-before relation and locksets, which they called "lock covers." Their subtraction optimization uses a notion similar to the weaker-than relation, but they only suggest using the optimization in the detector itself, while we employ the notion in many stages of our detection framework.

Eraser is similar to our approach in that its datarace detection algorithm is based on lock-based synchronization [24]. However, Eraser enforces the constraint that each shared memory location is protected by a unique lock throughout an execution, which we do not, thus reporting fewer spurious data races. Our ownership model is based on Eraser's, but Eraser has no comparable handling of the `join` operation (see Section 8). Eraser works independently of the input source language by instrumenting binary code, but its runtime overhead is in the range of 10× to 30×.

Praun and Gross's *object race detection* [21] greatly improves on Eraser's performance by applying escape analysis to filter out non-datarace statements and by detecting dataraces at the object level instead of at the level of each memory location (their overhead ranges from 16% to 129% on the same benchmarks we used, with less than 25% space overhead). However, their coarser granularity of datarace detection (which includes treating a method call on an object as a write) leads to the reporting of many dataraces which are not true dataraces, *i.e.,* the reported races do not indicate unordered concurrent accesses to shared state. For example, on the `hedc` program, we report dataraces on 5 objects, all of which are true dataraces, while object race detection reports over 100 dataraces, almost all of which are not true dataraces. (The race definitions for object race detection and Eraser imply they always report a superset of the races we report.)

TRaDe is similar to object race detection in that they both apply escape analysis [13], although TRaDe does the analysis dynamically. TRaDe's datarace detection differs from ours in that it is based on the happens-before relation. TRaDe adds a runtime overhead ranging from 4× to 15× [13] compared to an interpreter, with approximately 3× space overhead. AssureJ [18] and JProbe [17] are commercial products that can dynamically detect dataraces in Java programs. AssureJ has been observed to have overhead ranging from 3× to 30×, while JProbe's memory requirements make its use practically impossible for any reasonably sized program [13].

Min and Choi's hardware-based scheme [19] uses the cache coherence protocol, and Richards and Larus' work [22] uses the *Distributed Shared-Memory (DSM)* computer's memory coherence protocol, respectively, in collecting information for on-the-fly datarace detection.

Most dynamic datarace detection techniques for SPMD programs work either as post-mortem tools or as on-the-fly tools [25], by collecting information from actual executions with software instrumentation. A post-mortem approach offers the possibility of improving on-line efficiency (by moving the bulk of the work to the post-mortem phase) at the cost of complicating ease-of-use. However, the size of the trace structure can grow prohibitively large thus making the post-mortem approach infeasible for long-running programs.

Another dimension that can be used to classify past work on datarace detection is the underlying *concurrency model*. Past work on datarace detection was historically targeted to multithreaded *fork-join* programs [1, 8]. However, those results are not applicable to the object-based concurrency models present in multithreaded *object-oriented* programming languages such as Java.

Netzer and Miller categorize dynamic dataraces into *actual*, *apparent*, and *feasible* dataraces [20]. Assuming `T10:this` and `T20:q` in Figure 2 point to different synchronization objects, `T11` and `T21` are both an actual and a feasible datarace if `T20` occurs before `T13`. They are, however, only a feasible datarace if `T13` occurs before `T20`, which introduces a happened-before relation from `T11` to `T21`.

Choi and Min describe how to identify and reproduce the *race frontier*, which is the set of dataraces not affected by any other dataraces [12]. By repeatedly reproducing and correcting the dataraces in the race frontier, one can identify all the dataraces that occur in executions.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach to efficient and precise datarace detection for multithreaded object-oriented programs. Our approach consists of a unique combination of static datarace analysis, optimized instrumentation, runtime access caching and runtime detection phases. This approach results in a runtime overhead that is only in the 13% to 42% range, well below most past work. Furthermore, our datarace definition is precise enough that in our test cases, almost all the dataraces reported were in fact concurrent accesses to shared memory locations without any ordering constraints. These results show that it is feasible to perform precise datarace detection in a production setting.

In the future, we plan to broaden the static/dynamic coanalysis approach to tackle other problems such as deadlock detection and immutability analysis. We also intend to enhance the static analysis phases with more precise alias analysis algorithms. We plan to integrate these new analyses with the record/replay capabilities of our DejaVu debugger [9], providing a powerful platform for reasoning about the behavior of multithreaded programs.

Acknowledgments

We would like to thank members of the Jikes RVM runtime group and the Jikes RVM optimization group at IBM T. J. Watson Research Center for their help with the Jikes RVM system. We also thank the referees and the committee members of PLDI for their insightful comments.

We thank Julian Dolby for his GNOSIS interprocedural analysis framework, which forms the basis of our static datarace analysis.

11. REFERENCES

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*, pages 342–354, January 1998.

[2] B. Alpern, et.al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of java without data races. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[4] B. Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

[5] J. Bodga and U. Hölzle. Removing unnecessay synchronization in Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[7] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *7th International Workshop on Languages and Compilers for Parallel Computing*, 1994. Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September, 1994.

[8] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[9] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th IEEE International Parallel & Distributed Processing Symposium*, April 2001.

[10] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[11] J.-D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research, 2001. Report RC22146; www.research.ibm.com/jalapeno/dejavu/.

[12] J.-D. Choi and S. L. Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[13] M. Christiaens and K. De Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. *Proceedings of the Java Virtual Machine Rsearch and Technology Symposium (JVM'01)*, April 2001.

[14] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):85–96, 1991.

[15] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, June 2000.

[16] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[17] KL Group, 260 King Street East, Toronto, Ontario, Canada. Getting Started with JProbe.

[18] Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7345, USA. AssureJ User's Manual, 2.0 Edition, March 1999.

[19] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1991.

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

[21] C. v. Praun and T. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[22] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 40–47, August 1998.

[23] E. Ruf. Effective synchronzation removal for Java. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[25] E. Schonberg. On-The-Fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297, June 1989.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *In Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41, January 1996.

[27] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.