Lightweight Data Race Detection for Production Runs *

Swarnendu Biswas
University of Texas at Austin (USA)
sbiswas@ices.utexas.edu

Man Cao
Ohio State University (USA)
caoma@cse.ohio-state.edu

Minjia Zhang
Microsoft Research (USA)
minjiaz@microsoft.com

Michael D. Bond
Ohio State University (USA)
mikebond@cse.ohio-state.edu

Benjamin P. Wood
Wellesley College (USA)
bpw@cs.wellesley.edu

Abstract

To detect data races that harm production systems, program analysis must target production runs. However, sound and precise data race detection adds too much run-time overhead for use in production systems. Even existing approaches that provide soundness *or* precision incur significant limitations.

This work addresses the need for soundness (no missed races) and precision (no false races) by introducing novel, efficient production-time analyses that address each need separately. (1) *Precise* data race detection is useful for developers, who want to fix bugs but loathe false positives. We introduce a precise analysis called *RaceChaser* that provides low, bounded run-time overhead. (2) *Sound* race detection benefits analyses and tools whose correctness relies on knowledge of *all* potential data races. We present a sound, efficient approach called *Caper* that combines static and dynamic analysis to catch all data races in observed runs. RaceChaser and Caper are useful not only on their own; we introduce a framework that combines these analyses, using Caper as a sound filter for precise data race detection by RaceChaser.

Our evaluation shows that RaceChaser and Caper are efficient and effective, and compare favorably with existing state-of-the-art approaches. These results suggest that RaceChaser and Caper enable practical data race detection that is precise and sound, respectively, ultimately leading to more reliable software systems.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors, testing tools; D.3.4 [*Programming Languages*]: Processors—compilers, debuggers, run-time environments

Keywords Data races; dynamic analysis; escape analysis; sampling

1. Introduction

In a multithreaded, shared-memory program execution, a *data race* occurs when two accesses are *conflicting* (two threads access the same variable with at least one write) and *concurrent* (not ordered by synchronization operations) [2]. Data races often directly or indirectly lead to concurrency bugs [42, 55]. The presence of data races—whether accidental or intentional—can affect an execution by crashing, hanging, or silently corrupting data [32, 36, 51]. Data races were culprits in the Therac-25 disaster [40], the Northeastern electricity blackout of 2003 [66], and the mismatched NASDAQ Facebook share prices of 2012 [56]. Data races will only become *more* problematic as software systems become increasingly parallel to scale with parallel hardware.

Data races not only tend to be associated with bugs and errors, but they lead to weak or undefined semantics for modern shared-memory languages and systems [1, 12, 13, 45]. For example, a Java or C++ program with ostensibly "benign" data races can behave erroneously, as a result of compiler transformations or being ported to a different architecture [11, 14, 62].

Detecting data races. The wide-ranging consequences of data races make it imperative to detect as many races[1] as possible. Program analyses can detect data races, but there exists a fundamental tradeoff between *soundness* (no missed races) and *precision* (no false race reports). Most static and some dynamic analyses provide soundness[2] but report many false races [20, 23, 27, 49, 50, 53, 54, 58, 59, 68, 69, 72], which developers find unacceptable [6, 46]. On the other hand, precise dynamic analyses generally cannot detect races in executions *other than* the current execution [9, 15, 28, 31, 37, 46]. This tradeoff is compounded by a second challenge: the *occurrence* of a data race is sensitive to thread interleavings, program inputs, and execution environments—so data races can remain undetected during in-house testing, even for extensively tested programs [66], and occur unexpectedly in production runs [40, 56, 66].

To find data races that manifest only in production, data race detection *must target* production runs. However, in practice, data race detectors see little use in production runs, due to high run-time overheads [46]. This paper addresses an open challenge: devising *sound and precise* data race detection analyses that are *efficient* enough for production systems.

Our approach. A key, motivating insight of this work is that although sound and precise data race detection is too inefficient for production, *separate analyses that each provide soundness or precision are beneficial*. Detecting precise (real) data races enables developers to find and fix software bugs, ultimately improving software reliability. Detecting a sound overapproximation of data races can help simplify and optimize dynamic analyses and runtime systems such as record & replay [39], atomicity checking [30], and software transactional memory systems [64]. Soundness and pre-

* This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, CCF-1421612, and XPS-1629126.

CC'17, February 5–6, 2017, Austin, TX, USA
ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033020

[1] In the rest of the paper, "race" always means "data race."

[2] Following prior work, we consider a dynamic analysis to be *sound* if it misses no true races that exist in *observed* executions.

cision are not only orthogonal but also complementary: the results of sound race detection can help simplify the work of precise race detection. Section 2 further motivates these benefits.

Mirroring this idea of separating precision and soundness, *our approach decouples data race detection as two complementary, lightweight analyses that maintain and refine a precise underapproximation and a sound overapproximation of all data races over the course of many production runs.*

To get a precise underapproximation (i.e., can miss data races), we introduce a novel dynamic analysis called *RaceChaser* (Section 4). Each run of RaceChaser takes one potential data race as input, and tries to detect whether the potential race occurs in the run. RaceChaser provides low, *bounded* run-time overhead, and it does not rely on any hardware support. RaceChaser is the first approach to adapt collision analysis to the paradigm of detecting one potential race per run. Our results suggest that RaceChaser outperforms the closest related work, which adds unbounded, unscalable run-time overhead [37] (Sections 2.1 and 8.2.1).

For a sound overapproximation of data races from observed runs, we introduce a novel approach called *Caper* (Section 5). Caper captures the set of all potentially racing pairs of accesses to escaped memory locations. Each production run under Caper targets only potential races identified by a sound (no missed races) static analysis, but not yet observed in prior runs. Caper provides soundness over observed runs, notably better precision than static analysis, and low performance overhead (Section 8.3). This balance provides a better precision–performance tradeoff than prior work (Sections 2 and 9).

In addition to the individual benefits of the two analyses, they can be integrated in a data race detection framework such that the precise RaceChaser analysis uses the potential data races identified by the sound Caper analysis (Section 6).

We have implemented RaceChaser and Caper in a high-performance Java virtual machine [3] (Section 7). Our evaluation compares RaceChaser and Caper empirically with closely related work (Section 8). Overall, RaceChaser provides better performance (lower, scalable, bounded overhead) and the same race coverage as prior work. Caper is significantly more precise than static analysis, and adds low enough overhead for production runs, unlike other known approaches that are dynamically sound (no missed races in observed executions). These results suggest that RaceChaser and Caper can be employed continuously in production settings.

This work advances the state of the art by demonstrating two analyses, which can be used on their own or in cooperation, that each provides a better performance–accuracy tradeoff than existing approaches. By using our analyses in production systems, this work has the potential to detect hard-to-catch data races and ultimately lead to more robust software systems.

2. Background and Motivation

Data race detection that is *either* precise or sound has distinct benefits. However, prior approaches that are precise or sound have serious drawbacks for use in production.

2.1 Detecting Real Data Races Only

Most state-of-the-art data race detectors track the *happens-before* relation [38] to detect conflicting accesses that are unordered by happens-before [26, 57]. However, happens-before analysis adds substantial run-time overhead at each variable access and synchronization operation (e.g., $>8\times$ slowdown [31]).

To target production environments, where the key constraint is run-time overhead, *sampling-based* race detection analysis trades coverage for lower overhead [15, 17, 28, 37, 46]. However, sampling approaches suffer from run-time overhead that is still high, unscalable, and unbounded. *LiteRace* and *Pacer* sample race detection analysis but instrument all program accesses, adding high baseline overhead even when the sampling rate is miniscule [15, 46]. *RaceMob* applies sampling by limiting its analysis to a single pair of static accesses per execution, limiting its instrumentation overhead [37]—but nonetheless it incurs high, unscalable, unbounded overhead, as we show empirically in Section 8.2.1.

While most sampling-based approaches track the happens-before relation, *DataCollider* exposes and detects simultaneous, conflicting accesses—a sufficient condition for a data race [28]. Our paper refers to this kind of analysis as *collision analysis*. (Prior work has also employed collision analysis in a non-sampling context [60].) To expose and detect simultaneous, conflicting accesses, DataCollider periodically pauses a thread's execution at a potentially racy access; in the meantime, other threads detect conflicting accesses to the same variable. DataCollider avoids heavyweight instrumentation by using hardware debug registers to monitor memory locations [28]. However, debug registers are hardware specific and thus unportable; architectures often have only a few debug registers, limiting the number of memory locations that can be monitored simultaneously. The overhead of setting debug watchpoints with inter-processor interrupts increases with the number of cores, so DataCollider may not scale well to many cores [67]. Furthermore, although developers or users can adjust DataCollider's run-time overhead by adjusting the sampling rate, it does not guarantee bounded run-time overhead.

CRSampler infers data races by detecting conflicts between concurrently executing regions [17], similar to existing region conflict analyses, particularly *Valor* [9] (Section 9). Like DataCollider, CRSampler performs efficient sampling by using hardware debug registers.

In summary, existing precise data race detection analyses are ill suited to production settings. This work asks the question: what is the best use of the production setting for detecting data races precisely? In particular, *is it possible to design a precise, portable analysis that adds low, bounded overhead?*

2.2 Detecting All Data Races in Observed Executions

While sampling-based approaches are precise, they are inherently unsound, missing data races that occur in production runs. This situation is unacceptable for analyses and systems whose correctness relies on soundly knowing *all* data races, such as record & replay systems [39], atomicity violation detectors [8, 30], and software transactional memory [64]. For example, *Chimera* provides sound multithreaded record & replay by conservatively tracking ordering between all potential data races, as identified by sound static analysis [39]. Although whole-program static analysis can provide a sound set of potential data races, its precision does not scale with program size and complexity, and it suffers from many false positives [27, 49, 50, 58, 69]. Chimera (and other approaches) could provide significantly better performance by using a more precise set of potential data races. In addition to benefiting unrelated analyses and systems, sound race detection can simplify or optimize *precise* race detection by providing a set of potential data races from observed executions (Section 6).

This work asks the question: *is it possible to exploit the production setting in order to detect all data races soundly, i.e., without missing any true races that have occurred in production runs?* To be useful, such an approach must add minimal overhead and provide *reasonable* precision that is significantly better than that of static analysis. To our knowledge, prior work provides no solution that is simultaneously sound, more precise than static analysis, and efficient enough for always-on use in production environments.

3. Overview

This paper addresses the two challenges motivated by the previous section, by introducing two complementary, lightweight production-time analyses. Section 4 presents *RaceChaser*, a precise analysis that detects *only true* data races. Section 5 presents *Caper*, a sound approach that detects *all* data races in observed runs. These two approaches—one precise but unsound, the other sound but imprecise—are inherently complementary (Section 6). They maintain and refine an underapproximation and overapproximation, respectively, of the set of real data races in observed executions. Furthermore, the two techniques can be integrated in a data race detection framework in which the precise analysis uses the overapproximation produced by the sound analysis.

4. RaceChaser: Precise Data Race Detection

This section describes a new analysis called *RaceChaser* that targets production-time race detection with minimal overhead. RaceChaser limits its analysis to a single potential data race per program execution, and it bounds the run-time overhead it adds. In practice, RaceChaser could get each potential race from a set of races identified by developers or by another analysis that identifies potential races [37, 60]—including our Caper analysis, as Section 6 describes. While it may seem unsatisfying to try to detect only one potential race per execution, we note that current practice is to perform *no* race detection in production!

RaceChaser provides several properties aside from precision: bounded time and space overhead, scalability (i.e., time and space overheads remain stable with more threads), and portability. Like DataCollider [28], RaceChaser employs collision analysis. However, DataCollider (and CRSampler [17]) cannot bound their overhead, nor are they portable (Section 2.1). On the other hand, sampling-based analyses that track happens-before provide precision and portability, but cannot provide low, scalable, or bounded overhead [15, 37, 46] (Section 2.1).

4.1 How the Analysis Works

A production run executing RaceChaser takes as input a single *potential* data race, which is an ordered pair of program sites, $\langle s_1, s_2 \rangle$. A *site* is a unique static program location (e.g., a method and bytecode index in Java). RaceChaser limits its analysis to these two sites (or one site if $s_1 = s_2$). When $s_1 \neq s_2$, RaceChaser considers $\langle s_1, s_2 \rangle$ and $\langle s_2, s_1 \rangle$ to be distinct, and it uses separate runs to try to detect them.

Before a thread T executes an access to memory location m at s_1, the analysis potentially *samples* the current access by *waiting*, i.e., pausing the current thread for some time. The analysis updates global analysis state to indicate that T is waiting at s_1 to access m.

When a thread T' executes an access to memory location m at s_2, the analysis checks whether some thread T is already waiting at s_1 before accessing the same location m. If so, the analysis has detected a true data race, and it reports the call stacks of T and T'.

Aside from the cost of waiting at some instances of s_1, the analysis can achieve very low overhead because it instruments only two (or one, if $s_1 = s_2$) static sites and—like other collision analyses [17, 28, 60]—avoids instrumenting synchronization operations. Although updating global analysis state could be a scalability bottleneck, these updates occur when at least one thread is waiting—and the fraction of time spent waiting is bounded, as discussed next.

Instrumentation overhead. Although RaceChaser bounds how long it spends waiting, it always executes instrumentation at s_1 and s_2, which is lightweight in the common case (particularly because it uses optimizations described shortly) but could incur nontrivial overhead for very frequent accesses. RaceChaser could control this cost at run time by detecting very frequent accesses (e.g., if s_1 or s_2 is in a hot, tight loop) and triggering dynamic recompilation or code patching to remove the instrumentation.

4.2 Sampling Policy

RaceChaser's sampling policy bounds the total amount of time spent waiting, and it prioritizes instances of s_1 more likely to be involved in data races.

Budgeting the overhead of waiting. RaceChaser is a "best-effort" approach that seeks to detect data races without impacting production performance significantly. As in prior work called *QVM* that targets a specified maximum overhead by throttling the analysis [4], RaceChaser targets a maximum overhead r_{max} specified by developers or users. To enforce this maximum, RaceChaser keeps track throughout execution of (1) wall-clock time of the *ongoing* execution, t_{total}, and (2) wall-clock time *so far* during which one or more threads have waited, t_{waited}, taking into account overlapped waits. Throughout an execution, RaceChaser ensures the following:

$$\frac{t_{waited}}{t_{total}} \leq r_{max}$$

This condition is conservative because a thread waiting for time t does not necessarily extend total execution time by t.

RaceChaser computes the following probability for whether to take a sample at s_1:[3]

$$P_{budget} = 1 - \frac{t_{waited} + t_{delay}}{(t_{total} + t_{delay}) \times r_{max}}$$

This computation is based on the fraction of time that will have been spent waiting *after* waiting for a planned amount of time t_{delay}. Note that P_{budget} will be close to 0 if RaceChaser is near its maximum overhead. P_{budget} will be close to 1 if RaceChaser is substantially under-budget.

Prioritizing accesses. Some static sites execute only once per run, while others execute millions of times. To account for this uncertainty and variability, it is important to wait at the first instance of a site (it might be the only one!) but less important to wait at later instances. This reasoning follows the intuition behind the *cold-region hypothesis*, which postulates that the likelihood of detecting bugs in some part of a program is inversely proportional to the execution frequency of that part of the program [19, 46]. As in prior work on sampling-based data race detection [46], we find that it is important to prioritize sampling of each *thread*'s initial access(es) (instead of the global execution's initial accesses) at s_1.

RaceChaser thus samples an instance of s_1 at a rate inversely proportional to $\mathit{freq}(s_1, T)$, the execution frequency of s_1 by the current thread T so far:

$$P_{\mathit{freq}} = \frac{1}{\mathit{freq}(s_1, T)}$$

RaceChaser can compute $\mathit{freq}(s_1, T)$ by counting how many times thread T has executed s_1 or how many times T has sampled s_1. Our implementation and evaluation use the latter policy.

Overall sampling probability. RaceChaser's sampling policy combines P_{budget} and P_{freq}:

$$P_{RaceChaser} = P_{budget} \times P_{\mathit{freq}}$$

A remaining issue is that soon after an execution starts, RaceChaser will have virtually no budget ($P_{budget} \approx 0$), so very early accesses will go unsampled, potentially missing some data races consistently. To detect such data races, it seems unavoidable that an execution must risk exceeding its budget. RaceChaser assumes a minimum total running time t_{min} and uses $max(t_{min}, t_{total})$ in place of t_{total} when computing P_{budget}. Developers can estimate a conservative value for t_{min} using their knowledge of the program and likely execution scenarios. In our experiments, t_{min} = 1 second.

The probability function $P_{RaceChaser}$ is *one of many possible* functions that satisfy our goals of (1) staying under an overhead budget and (2) prioritizing each thread's early accesses. It may suffer drawbacks, e.g., it may sample too infrequently as $\mathit{freq}(s, T)$

[3] Although RaceChaser could choose which dynamic accesses to sample using a deterministic function, it instead uses *randomness*, waiting at each dynamic access with some probability P as described. Using randomness increases the chances of finding a race in the long run, over multiple executions, although in practice RaceChaser detects most races consistently from run to run.

grows, significantly undershooting the overhead budget. That said, we have found the function to be suitable in practice for achieving RaceChaser's design goals.

4.3 Optimizations

RaceChaser's low instrumentation overhead and good scalability (Section 8.2) rely on the following optimizations.

Sampling check. RaceChaser's instrumentation at s_1 uses an optimized, two-part check, which is equivalent to sampling with probability $P_{RaceChaser}$. The instrumentation first computes P_{freq}, which is thread local and cheap. Then, with probability P_{freq}—that is, infrequently in the common case—the instrumentation computes P_{budget} and samples the access with probability P_{budget}.

Avoiding scalability bottlenecks. Whenever instrumentation at s_1 waits, it acquires a lock on global analysis state, records that the current thread T is waiting at site s_1 on memory location m, and increments a global count of waiting threads. The instrumentation then performs a non-busy timed wait on the global lock, which atomically releases the lock while waiting. When the timed wait completes, the instrumentation atomically reacquires the global lock, restores global analysis state to indicate that T is no longer waiting, and releases the global lock. A thread tries to acquire the global lock at s_1 only when it should wait, so any lock contention costs are effectively bounded.

Instrumentation at s_2 checks if another thread is waiting on the same memory location m at s_1. To do this efficiently, s_2 first checks the global counter to see if any threads are waiting, *avoiding* remote cache misses in the common case when no threads are waiting. It acquires the global lock and accesses global analysis state only if at least one thread is waiting at s_1, thus limiting the possibility of lock thrashing in the common, non-waiting case. We implement the counter as a Java volatile variable, so this optimization does not introduce any unsoundness.

5. Caper: Detecting All Potential Data Races

This section focuses on the problem of detecting a sound overapproximation of *all* data races that occur in production runs. Although whole-program static analysis can provide a sound set of potential races, it reports *many* false races (Section 8.3). Current predictive race detection approaches do not scale to large programs [35, 65]. Can a sound, low-overhead approach provide significantly better precision than static analysis alone?

5.1 Caper Overview

We introduce a novel analysis called *Caper* that detects a set of potential data races that includes all true data races from observed executions. Caper combines static and dynamic analyses. Caper initially runs a static analysis (e.g., [27, 49, 50, 58, 69]) to produce a set of *statically possible* pairs of racy accesses, $spPairs$. This set consists of unordered pairs $\langle s_1, s_2\rangle$ (i.e., $\langle s_1, s_2\rangle = \langle s_2, s_1\rangle$) where s_1 and s_2 are each a static program location. The set $spPairs$ need not be particularly precise—and in fact we find that a state-of-the-art static analysis [50] reports many false races (Section 8.3).

During each program execution, Caper's *dynamic* analysis identifies dynamically possible race pairs, $dpPairs$, from $spPairs$. Each invocation of Caper moves newly identified possible race pairs from $spPairs$ to $dpPairs$. At any time, $spPairs \cap dpPairs = \emptyset$, and $dpPairs$ is a sound overapproximation (no missed races) of every data race that has occurred across all analyzed executions.

Caper incurs low run-time overhead by using two insights. First, Caper refines $spPairs$ and $dpPairs$ on each successive program execution, so that in steady state, production runs are unlikely to detect and move any new pairs from $spPairs$ to $dpPairs$. This feature allows Caper to optimize for *not* detecting new pairs in $spPairs$ that should move to $dpPairs$. Second, Caper employs lightweight, sound dynamic escape analysis, described shortly.

Although Caper's run-time overhead in steady state is low, it is *not* bounded (unlike RaceChaser). To provide soundness, it seems infeasible to bound overhead (without "giving up" and declaring all pairs in $spPairs$ to be in $dpPairs$).

5.2 Caper's Dynamic Analysis

Caper's dynamic analysis builds on *dynamic escape analysis* (DEA), which identifies objects that have potentially become shared (accessed by two or more threads). Caper's DEA instruments all accesses to track the escape property soundly, while Caper's sharing analysis instruments sites in $spPairs$, i.e., every site s such that $\exists s' \mid \langle s, s'\rangle \in spPairs$. Instrumentation at s checks if the object accessed at s has escaped; if so, it marks s as "escaped." Caper maintains a set of dynamically escaped sites called $deSites$ such that

$$deSites = \big\{ s \mid (\exists s' \mid \langle s, s'\rangle \in spPairs \cup dpPairs) \land \\ s \text{ escaped in an analyzed execution} \big\}$$

Caper removes $\langle s_1, s_2\rangle$ from $spPairs$ and adds it to $dpPairs$ for future invocations of Caper if both s_1 and s_2 are in $deSites$, i.e., $dpPairs = \{\langle s_1, s_2\rangle \mid s_1 \in deSites \land s_2 \in deSites\}$. Future runs of Caper instrument only the sites that are in $spPairs$, but DEA continues to monitor *all* memory accesses.

Caper's reachability-based DEA. Caper's form of DEA is based on the idea that an object may have escaped if it is transitively reachable from another escaped object. *Reachability-based DEA* conservatively identifies the first time an object becomes reachable by some thread other than its allocating thread, using these rules:

- Each newly allocated object starts in the NOT_ESCAPED state. Thread objects (e.g., java.lang.Thread objects in Java) are initialized to the ESCAPED state.
- At each reference-type store to a global variable (C.sf = p), the object referenced by p becomes ESCAPED.
- At each reference-type store to an instance field (q.f = p), the object referenced by p becomes ESCAPED if the object referenced by q is ESCAPED.
- Whenever an object becomes ESCAPED, all objects transitively reachable from the object become ESCAPED.

Soundness. Importantly, this reachability-based DEA *soundly* detects *shortest* data races, which are data races that are *not* dependent on some other race. In contrast, a non-shortest data race r "depends" on some other data race r', meaning that eliminating r' necessarily eliminates r. Consider the following example:

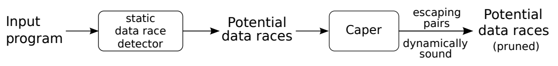
```
// T1:                          // T2:
x.f = ...;  // s1
C.sf = x;   // s2
                                y = C.sf;   // s3
                                 ... = y.f; // s4
```

Suppose thread T1's local variable x initially refers to an unescaped object o. Object o becomes reachable by other threads when T1 publishes a reference to it in the escaped field, C.sf.[4] T2 acquires this reference via C.sf and uses it to access o's field f.

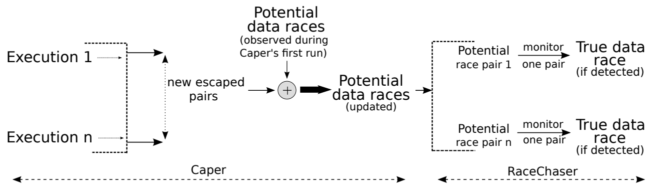
T1 and T2 race on C.sf at s_2 and s_3. This is a shortest race: it does not depend on any other races. T1 and T2 also perform unsynchronized accesses to o.f at s_1 and s_4, but this race depends on the other, "shorter" race. s_1 and s_4 can only race on o.f if s_2 and s_3 also race on C.sf. Fixing the s_2–s_3 race (e.g., by making C.sf a volatile field in Java) necessarily fixes the s_1–s_4 race.

Caper's escape analysis targets memory- and type-safe languages in order to compute dynamic reachability-based escape accurately. In contrast to Caper, prior DEA-based data race detec-

[4] The example uses a static field for simplicity. Any escaped field suffices.



(a) Caper's first run, usually during testing.



Execution 1
Execution n
new escaped pairs
Potential data races (observed during Caper's first run)
Potential data races (updated)
Potential race pair 1
monitor one pair
True data race (if detected)
Potential race pair n
monitor one pair
True data race (if detected)
Caper
RaceChaser

(b) Caper and RaceChaser's runs during production.

Figure 1. Runs of Caper and RaceChaser can work together to detect data races efficiently during production.

tion is generally unsound [21, 33, 41, 53, 54, 59] (Section 9). To our knowledge, our work is the first to apply a form of DEA as a fully sound filter for data race detection. More significantly, to our knowledge, Caper is the first to use DEA to prune the results of static data race detection.

As presented, Caper's soundness argument does not apply to escaping `memory_order_relaxed` accesses in C++. For the above example, if s_2 and s_3 were `memory_order_relaxed`, Caper would miss the s_1–s_4 race, but the s_2–s_3 pair would no longer be considered a race. The underlying issue is that the Java and C++ memory models define data races differently [13, 45]. Caper could be extended to warn separately about `memory_order_relaxed` escape. Alternatively, since Caper is already imprecise, it could treat `memory_order_relaxed` accesses as normal accesses that may race.

Balance. Caper strikes a balance between soundness, precision, and performance that is unmatched by existing static and dynamic analyses. Compared to sound static analyses, Caper is significantly more precise in practice (Section 8.3). Compared to dynamic analyses that are precise for a single run, Caper provides soundness across all observed runs and has much lower run-time overhead than analyses that are sound and precise on a single run (Section 8.3.1). By observing all (or many) production runs, Caper has full (or high) coverage of all data races in production runs.

6. Synergy of RaceChaser and Caper

Detecting a sound overapproximation of data races in the current run by Caper allows its use as a dynamic filter for analyses that require a sound knowledge of all data races in the run, e.g., record & replay [39], atomicity checking [8, 30], and software transactional memory [64]. Furthermore, the sound-but-imprecise approach of Caper can provide a set of potential data races for the precise-but-unsound approach of RaceChaser.

Xie et al. observe that "data races should be detected with a range of tools used in stages, including both static and dynamic detectors" [72]. Figure 1 shows how the Caper and RaceChaser analyses can be staged to detect data races precisely and efficiently during production runs. Figure 1(a) shows the first execution of Caper during testing, when $dpPairs = \emptyset$ and $deSites = \emptyset$. During the testing runs, Caper will potentially find many escaped sites and can incur high overhead. Future invocations of Caper during production runs (Figure 1(b)) only monitor sites that are in $spPairs$. The low overheads of Caper in the steady state allow it to continually monitor program executions during production runs to find previously unseen potential data races, ensuring soundness across all production runs. The framework limits the the potential races monitored by RaceChaser to those reported by Caper.

We have not implemented this framework, nor evaluated Caper as the sole filter for RaceChaser. Although Caper is more precise than static analysis, it still reports many potential data races.

7. Implementation

We have implemented RaceChaser and Caper's dynamic analysis in Jikes RVM 3.1.3 [3], a Java virtual machine that has performance competitive with commercial JVMs [9]. We have also implemented an existing precise analysis for comparison with RaceChaser, and an existing sound analysis for comparison with Caper. We have made our implementations publicly available on the Jikes RVM Research Archive.[5]

7.1 Detecting Data Races Precisely

RaceChaser. RaceChaser's implementation closely follows Section 4's design. Although RaceChaser could dynamically remove the instrumentation at s_1 and s_2 when these sites are very frequent (e.g., in a hot, tight loop; Section 4.1), we have not implemented this feature since it is not needed for our experiments.

Instrumenting different accesses in each production run, as in RaceChaser, naturally lends itself to just-in-time-compiled languages such as Java: the dynamic compiler instruments only those accesses targeted by the current run. An implementation in an ahead-of-time compiled language such as C++ could use dynamic code patching, distribute a different binary to each production site, or (at higher cost) instrument all potentially racy sites [37].

LiteHB. We have implemented an analysis that we call *LiteHB* that is based on RaceMob's happens-before analysis [37].[6] Like RaceChaser, LiteHB takes as input a single potential data race $\langle s_1, s_2 \rangle$, and tries to detect it in the current execution.

LiteHB implements RaceMob's optimized tracking of the happens-before relationship [37]. In particular, LiteHB (1) only starts tracking happens-before after some thread executes s_1 and (2) stops tracking happens-before when all threads can no longer race with a prior instance of s_1, i.e., when all threads' vector clocks *happen after* the clocks for all executed instances of s_1.

Our LiteHB implementation adds the following metadata to each object: a header word to track the vector clock if the object's lock is acquired; a header word that points to an array of metadata, which is used if the object is an array that is accessed by LiteHB's instrumentation; and a word of metadata for each field, which is used if the field is accessed by LiteHB's instrumentation. With additional engineering effort, LiteHB could avoid adding some of this metadata; in particular, it could limit per-field metadata to those fields that s_1 and s_2 might access. Instead, our evaluation of LiteHB accounts for the costs of extraneous metadata.

7.2 Detecting Potential Data Races Soundly

Caper. As described in Section 5, Caper starts with statically overapproximating the set of potential data races. For Caper's static analysis, we use the publicly available static data race detector *Chord*, revision 1825 [50]. Chord's default setup is *unsound* because it uses a may-alias analysis for locks.[7] We thus *disable* Chord's static lockset analysis, along with other unsound options such as ignoring accesses in constructors, and we enable a Chord feature that resolves reflective calls by executing the program. The result is a sound analysis that identifies potential races based solely on (static) thread escape analysis and fork–join analysis. Although Chord analyzes each program together with the Java libraries, it analyzes a different library implementation than what Jikes RVM uses, so our experiments detect potential races in application code only. Caper uses the set reported by Chord as $spPairs$ (Section 5). Caper's dynamic analysis works as described in Section 5.

[5] `http://www.jikesrvm.org/Resources/ResearchArchive/`

[6] Although RaceMob's happens-before analysis also includes waiting at some accesses [37], we omit this feature from LiteHB in order to measure the cost of (optimized) happens-before analysis alone.

[7] We have confirmed with Naik that there is no available sound implementation of Chord that uses conditional must-not alias analysis [49].

Dynamic alias analysis. For comparison with Caper, we have implemented a *dynamic alias analysis*, which detects all pairs of aliasing sites. It reports $\langle s_1, s_2 \rangle$ if and only if accesses at s_1 and s_2 by threads T_1 and T_2 at memory locations m_1 and m_2 such that $T_1 \neq T_2 \wedge m_1 = m_2$.

Dynamic alias analysis is more precise than Caper but adds high run-time overhead (Section 8.3). (In contrast, RaceMob uses a form of dynamic alias analysis that checks one potentially aliasing pair in each execution [37].) Dynamic alias analysis offers a different point of comparison than static analysis, which is highly imprecise but adds no overhead. To our knowledge, there exists no sound analysis that has (1) better precision than Caper and (2) overhead low enough for production.

8. Evaluation

This section evaluates RaceChaser and Caper's efficiency and effectiveness, and compares with competing approaches.

8.1 Methodology

Benchmarks. Our evaluation analyzes and executes benchmarked versions of large, real applications. Our experiments execute the DaCapo benchmarks [10], versions 2006-10-MR2 and 9.12-bach, which we distinguish using names suffixed by 6 and 9; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[8] We omit programs with only one thread or that Jikes RVM cannot execute. We also omit eclipse6 since Chord's static analysis fails when analyzing it; and jython9 and pmd9 because Chord reports no potential data races for them.[9] We run the *large* workload size of the DaCapo benchmarks. Table 1(a) (page 8) reports total and maximum active threads for each program.

Platform. For each of our implementations, we build a high-performance configuration of Jikes RVM (FastAdaptive) that adaptively optimizes the application at run time and uses the default high-performance garbage collector, which adjusts the heap size automatically. The experiments execute on an AMD Opteron 6272 system with eight 8-core 2.0 GHz processors (64 cores total), running 64-bit RedHat Enterprise Linux 6.7, kernel 2.6.32.

8.2 Detecting Data Races Precisely

This section evaluates the run-time performance and race detection coverage of RaceChaser, compared with LiteHB.

Methodology. Each run of RaceChaser or LiteHB takes as input a potential data race $\langle s_1, s_2 \rangle$. In a real production setting, such potential races could be identified by developers or by another analysis. Potential races could be generated by a *sound* analysis such as static analysis (the approach taken by RaceMob [37]), testing-time analysis such as dynamic alias analysis, or our Caper production-time analysis (as Section 6 describes). However, to keep our experiments manageable (especially since we run multiple configurations and trials), we evaluate RaceChaser and LiteHB on a relatively small set of potential races: the set of access pairs identified by dynamic alias analysis *that also violate the dynamic lockset property,* i.e., the accesses hold no common lock while accessing the same variable. (Prior work introduces heavyweight dynamic analyses that check the lockset property [20, 23, 53, 54, 59, 68]. *RACEZ* samples memory accesses via the hardware *performance monitoring unit* (PMU), using imprecise lockset analysis and offline analysis of sampled accesses to maximize online performance [63]. Lockset analyses report false data races.) This methodology simulates a realistic scenario: *prioritizing* certain potential races by using a heavyweight *testing-time* analysis to prioritize a subset of potential races, albeit at the cost of missing a few true races. Although in general a prioritized set will miss some races, our experiments do not encounter this issue because they use the same inputs across configurations.

8.2.1 Performance

Run-time overhead. Figure 2 shows the overhead added over uninstrumented execution (unmodified Jikes RVM) for configurations of RaceChaser and LiteHB. Each bar is the mean execution time over 30 randomly selected potential races (from those identified by dynamic alias analysis that violate the lockset property). If a program has fewer than 30 potential races, we use multiple trials of each potential race.

All RaceChaser configurations use wait times of t_{delay} = 10 ms, which is sufficient to expose all data races that happens-before analysis detects in the evaluated programs. In general, increasing t_{delay} helps detect data races whose accesses are "far apart," but leads to fewer accesses being sampled.

The RaceChaser configurations are for three target maximum overheads, r_{max} = 0%, 5%, and 10%. The r_{max} = 0% configuration shows overhead without any waiting, measuring the cost of instrumentation at accesses alone. This overhead, which RaceChaser cannot measure or bound with its sampling model, is consistently low (always <5%). For r_{max} = 5% and 10%, RaceChaser stays under the target overhead across all programs, as expected. Since RaceChaser instruments only one pair of accesses, the instrumentation overhead is generally low. The average overheads for r_{max} = 5% and 10% are 1.0% and 1.6%, respectively. It is unsurprising that actual overheads are generally less than r_{max}, since RaceChaser's model conservatively assumes that pausing one thread by t_{delay} slows the entire program by t_{delay}.

The *LiteHB* and *FullHB* configurations are variants of LiteHB. *LiteHB* is the default LiteHB algorithm that uses RaceMob's optimized happens-before tracking; it adds 20% run-time overhead on average. *LiteHB (only HB tracking)* shows overhead incurred by *optimized* happens-before tracking only: it adds no instrumentation to accesses, except for instrumentation at s_1 that enables happens-before tracking if it is currently disabled. This configuration isolates the cost of optimized happens-before tracking, which incurs 13% average overhead.

We note that both LiteHB configurations include space overhead, as well as cache pressure and GC costs, from per-object and per-field metadata—but most of the per-field metadata could be avoided with additional engineering effort (Section 7.1). We find that per-field metadata *alone* adds run-time overhead of 7% (results not shown), so the true cost of optimized happens-before analysis is as low as 13%.

FullHB (only HB tracking) performs *unoptimized* happens-before tracking by performing vector clock computations at every synchronization operation; it adds no instrumentation at accesses. This configuration, which shows the benefit of RaceMob's happens-before optimization [37], adds 18% average overhead. The last configuration, *FullHB*, shows the overhead of instrumenting s_1 and s_2 and performing unoptimized happens-before tracking, and adds 37% overhead on average. These results show that RaceMob's optimization is indeed beneficial in our experiments.

For sunflow9, both LiteHB and FullHB add *especially high* overheads (162 and 526%, respectively). As the results indicate, most of this overhead actually comes from instrumentation at accesses, not from tracking happens-before. In sunflow9, instrumentation at mostly read-shared accesses must perform *updates* to per-variable clocks, leading to many remote cache misses. Such high, *unpredictable* overheads prohibit use of happens-before-analysis-based race detection on production systems.

Space overhead. We have also measured the *space* overhead incurred by RaceChaser and LiteHB (detailed results omitted). RaceChaser, which adds only a small amount of global metadata, adds 1% overhead on average and at most 5% overhead for any program (which is mostly due to increased dynamic compilation permit-

[8] `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

[9] pmd9 has data races [9], but Chord does not recognize its use of threads.

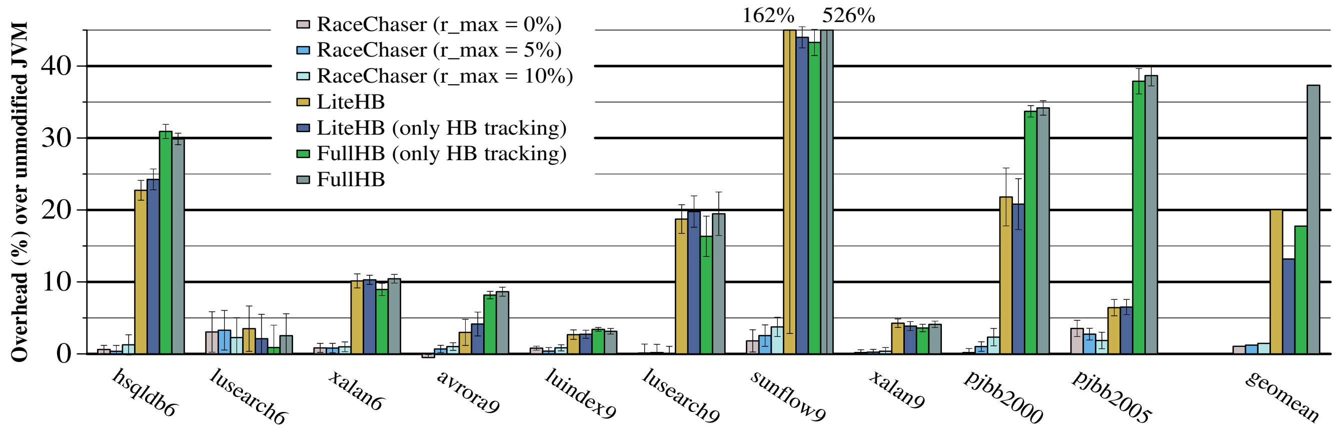


Figure 2. Run-time overhead of configurations of RaceChaser and LiteHB. The error bars represent 95% confidence intervals.

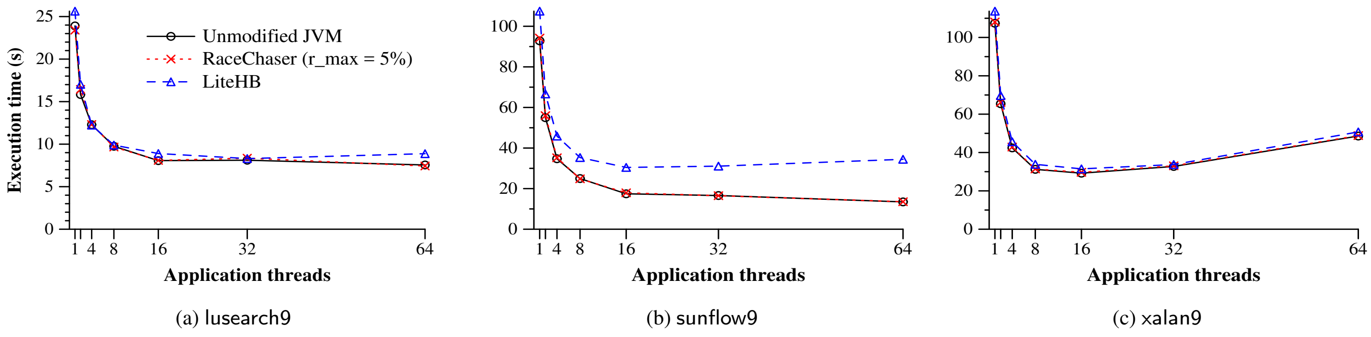


Figure 3. Execution times of RaceChaser (r_{max} = 5%) and LiteHB for 1–64 application threads. The execution times for 64 threads correspond to the results in Figure 2. The legend applies to all graphs.

ted by slower execution). For LiteHB, if we subtract out all space overhead due to per-object and per-field metadata (optimistically assuming all such metadata could be avoided; Section 7.1), space overhead is 5% on average and at most 29% for any program.

Scalability. Production software often has many threads and will likely have more as future systems provide more cores [43]. To compare how RaceChaser and LiteHB scale with more threads, we evaluate the three programs that support spawning a configurable number of application threads: lusearch9, sunflow9, and xalan9 (Table 1(a)). Figure 3 plots execution time for 1–64 application threads, using configurations from Figure 2. The *Unmodified JVM* configuration shows that lusearch9 and sunflow9 naturally scale with more threads, while xalan9 anti-scales starting at 16 threads.

RaceChaser (r_{max} = 5%) not only adds low overhead, but its overhead is largely unaffected by the number of threads for all programs. Note that in the plots, the *Unmodified JVM* and *RaceChaser* lines are difficult to distinguish. In contrast, for lusearch9 and sunflow9, *LiteHB* not only adds high overhead, but its overhead grows with more threads.[10] As discussed earlier, most of sunflow9's LiteHB overhead is due to conflicts from updating per-variable clock metadata—a cost that increases as the number of threads increases. Most of lusearch9's overhead comes from tracking happens-before via vector clock computations that take linear time in the number of threads. For xalan9, LiteHB always incurs very low overhead and has no noticeable scalability issues.

[10] For 32 threads, the *Unmodified JVM* and *RaceChaser* configurations experience anomalous performance for lusearch9. We have attributed this anomaly to Linux thread scheduling decisions [5].

These scalability results imply that LiteHB will add additional overhead for even larger thread counts, while RaceChaser will provide consistently low overhead. In general, LiteHB is afflicted with the drawbacks of happens-before analysis, which scales poorly with the number of application threads.

Comparison with RaceMob's reported results. Our RaceMob-like implementation of optimized happens-before analysis, evaluated on Java programs, adds 13% average overhead (after subtracting out overhead from per-field metadata). In contrast, Kasikci et al. report 2.3% average run-time overhead for RaceMob's happens-before analysis evaluated on C/C++ programs [37]. Although there are many implementation and experimental differences in play, we suspect three primary causes for the difference. First, our programs generally have more threads (Table 1). Second, Java programs tend to have much more frequent synchronization operations. Table 1 compares the sizes, measured in executed memory accesses, of synchronization-free regions for C/C++ programs and the Java programs that we evaluate, i.e., the ratio of total memory accesses to total synchronization operations. The C/C++ programs are the PARSEC benchmark suite [7], version 3.0-beta-20150206 (excluding freqmine since it uses OpenMP for its parallelization and facesim which does to run with our tool), with the *simmedium* input size; we count their synchronization operations and memory accesses by modifying a Pintool from prior work [22, 44]. With the exception of one Java program (sunflow9), synchronization operations are several orders of magnitude more frequent in Java programs than most C/C++ programs. A related issue is that the RaceMob authors report that their evaluated C/C++ programs execute synchronization barriers, which

Java program	Total	Max live	Avg. accesses per SFR
hsqldb6	402	102	26
lusearch6	65	65	156
xalan6	9	9	21
avrora9	27	27	553
luindex9	2	2	718
lusearch9	c	c	201
sunflow9	$2 \times c$	c	1,030,000
xalan9	c	c	53
pjbb2000	37	9	7
pjbb2005	9	9	15

(a) Java programs

C/C++ program	Threads	Avg. accesses per SFR n=8	n=16	n=32
blackscholes	$1+n$	9,150,000	4,750,000	2,290,000
bodytrack	$2+n$	63,600	57,400	47,800
canneal	$1+n$	5,470,000	2,746,000	1,370,000
dedup	$3+3n$	36,300	36,300	35,900
ferret	$3+4n$	630,000	514,000	388,000
fluidanimate	$1+n$	131	99	68
raytrace	$1+n$	5,820,000	3,030,000	1,550,000
streamcluster	$1+2n$	4,320	2,260	1,250
swaptions	$1+n$	83,000,000	41,600,000	20,800,000
vips	$3+n$	105,000	81,100	55,800
x264	$1+2 \times frames$	208,000	202,000	202,000

(b) C/C++ programs

Table 1. Spawned threads and average executed memory accesses per synchronization-free region (SFR), rounded to three significant figures and the nearest integer, for Java and C/C++ programs. (a) The table shows *Total* threads created and *Max live* at any time, which is dependent on the core count c (64 in our experiments) for three programs. (b) n is PARSEC's minimum threads parameter. $frames$ is the input-size-dependent number of frames processed by x264.

	RaceChaser		LiteHB		Overlap	
hsqldb6	10	(10)	10	(10)	10	(10)
lusearch6	0	(0)	0	(0)	0	(0)
xalan6	17	(17)	17	(17)	17	(17)
avrora9	7	(6)	7	(7)	7	(6)
luindex9	2	(2)	2	(2)	2	(2)
lusearch9	0	(0)	0	(0)	0	(0)
sunflow9	2	(2)	2	(2)	2	(2)
xalan9	7	(7)	7	(7)	7	(7)
pjbb2000	7	(7)	7	(7)	7	(7)
pjbb2005	1	(1)	1	(1)	1	(1)
Total	53	(52)	53	(53)	53	(52)

Table 2. Data races reported by RaceChaser and LiteHB. The two numbers are distinct races reported at least once and (in parentheses) at least twice, out of 10 trials. *Overlap* is distinct races reported by both analyses.

RaceMob-optimized happens-before analysis automatically exploits by halting tracking of happens-before immediately after a barrier [37]. In contrast, the Java programs we evaluate execute few or no synchronization barriers.

Comparison with other happens-before tracking. The state-of-the-art sound and precise data race detection analysis *FastTrack* incurs 750% overhead for the FastTrack authors' implementation and evaluation [31]. In prior work that implements FastTrack in Jikes RVM, it adds 340% overhead on average [9]. Industry-standard tools such as Intel's Inspector XE,[11] Google's ThreadSanitizer v2 [61], and Helgrind [52] are largely based on happens-before analysis. They add high run-time overheads and are suitable for testing runs only.

The sampling-based happens-before race detector *Pacer* avoids analysis at most operations, but still must instrument all operations [15]. For sampling rates of 0%, 1%, and 3%, Pacer implemented in Jikes RVM adds 33%, 52%, and 86% run-time overhead on average [15]—significantly higher than RaceChaser's overhead.

8.2.2 Detecting Real Data Races

Empirical comparison. Table 2 reports how many (true) data races RaceChaser and LiteHB detect. For each potential race pair from the sound set of lockset-violating access pairs, we execute 10 trials each of RaceChaser and LiteHB. The first column for each analysis reports distinct races reported at least once across 10 trials. To account for uncertainty about the repeatability of detecting a race reported in just one trial, we also report, in parentheses, races reported in at least two trials. We run RaceChaser configured with a target maximum run-time overhead of r_{max} = 5% and a fixed wait time of t_{delay} = 10 ms. The table omits 30 additional access pairs (all in sunflow9 or xalan9) reported by LiteHB, which are non-shortest, or dependent, races (Section 5.2).

The data races detected in these experiments are frequent races, i.e., they manifest commonly across multiple runs. RaceChaser detects all of the races detected by LiteHB, with the caveat that RaceChaser detects one race in avrora9 in only one of the trials. RaceChaser's per-trial coverage is dependent on its (randomized) sampling policy, which could benefit from further work. We have verified that the data races reported by RaceChaser and LiteHB match those reported by a publicly available implementation of FastTrack in Jikes RVM from prior work [9].

Despite using sampling-based collision analysis, RaceChaser provides essentially the same coverage as happens-before analysis for the evaluated programs. Furthermore, RaceChaser provides significantly lower, bounded overhead and better scalability than LiteHB.

8.3 Detecting Potential Data Races Soundly

This section evaluates the performance and precision of Caper, compared with static analysis and dynamic alias analysis.

8.3.1 Performance

Static analysis. Caper initially employs static analysis to generate the set $spPairs$. Static analysis needs to execute only once (or whenever the code changes) and does not affect run time, so its performance is not crucial. Chord (Section 7.2) takes at most 30 minutes to analyze any program.

Caper's dynamic analysis. Figure 4 shows the overhead added by Caper's dynamic analysis over unmodified Jikes RVM. Each bar is the mean of 25 trials, with 95% confidence intervals centered at the mean. The first bar, *DEA*, shows the overhead of reachability-based dynamic escape analysis alone, which incurs 3% overhead on average. *Caper (first run)* is Caper when it analyzes a program for the first time, i.e., when $dpPairs = \emptyset$ and $deSites = \emptyset$. Under these conditions, the analysis finds many newly escaped sites to add to $deSites$, incurring 27% average overhead.

In contrast, *Caper (steady state)*, represents performance during production, when prior testing (and production) runs have added nearly all escaped sites to $deSites$. Under these conditions, Caper's dynamic analysis elides instrumentation at sites in $deSites$, and it incurs little or no overhead from instrumented sites being added to $deSites$. On average, *Caper (steady state)* incurs 9% run-time overhead. These results suggest that Caper in steady state is efficient enough for many production environments.

The results for xalan9 are unintuitive. We find that adding *any* instrumentation to xalan9 *improves* performance relative to the

[11] https://software.intel.com/en-us/intel-inspector-xe

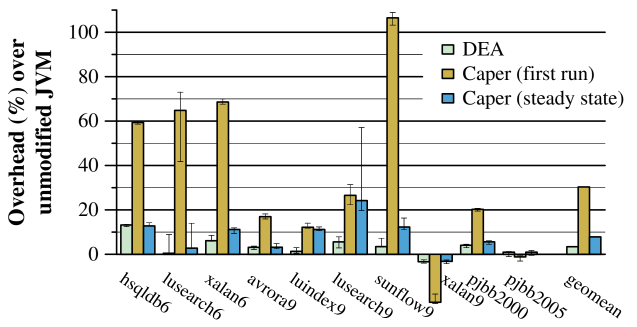


Figure 4. Run-time performance of Caper.

baseline. We believe this issue is caused by Linux thread scheduling decisions [5].

Dynamic alias analysis. We find that our implementation of dynamic alias analysis slows program execution by 13X on average (results not shown). Admittedly, this implementation is not heavily optimized; we have implemented it mainly for measuring its precision. But dynamic alias analysis is inherently a heavyweight analysis because it must keep track of *every* accessed combination of thread, site, and variable to detect future aliasing in the run.

8.3.2 Precision and Effectiveness

Precision. Table 3 shows potential races (unique, unordered access pairs) reported by static analysis (Chord), Caper, and dynamic alias analysis. The table counts a potential race if it is reported at least once across 10 trials. Although it is undecidable whether a potential race is real or not, we assume that the vast majority of potential races detected are false races; as evidence, we note that predictive analyses have been able to expose at most dozens, not thousands, of additional races beyond those found by precise dynamic analysis [29, 35, 65].

The *Static analysis* columns show potential race pairs identified by the unsound and sound versions of Chord. For most programs, the sound analysis reports tens of thousands of potential races. In comparison, Chord's *unsound* analysis, which is less imprecise since it uses Chord's may-alias lockset analysis [50], reports three times fewer potential races on average. However, the unsound analysis is demonstrably unsound: we find that it misses 17 of the 53 real data races identified by RaceChaser and LiteHB. We have verified that the other analyses (Chord's sound static analysis, Caper, and dynamic escape analysis) report all known true races.

The last two columns of Table 3 show the precision of dynamic approaches. Notably, Caper usually provides substantially better precision (often 1–2 orders of magnitude fewer potential races) than static analysis for all but two programs (avrora9 and pjbb2000). For the most part, dynamic alias analysis provides better precision than Caper, which is unsurprising since dynamic alias analysis detects aliasing and Caper detects reachability-based escaping. However, in some cases, Caper actually reports fewer potential races than dynamic alias analysis because Caper does not count a site as escaped if it accesses a non-escaped object, even if the object later becomes escaped. This effect is particularly pronounced for sunflow9, whose main thread initializes data before the data escapes and is accessed by worker threads. Importantly, *Caper's approach is sufficient both for detecting all true data races* (Section 5.2) *and for clients that need sound knowledge of data races or sharing [30, 39, 64]* (Section 2.2).

Effectiveness. To estimate the effectiveness of Caper in optimizing other dynamic analyses that must account for potential data races [30, 39, 64], we compute how many *dynamic (executed) accesses* are identified as being part of a potential data race. We expect this value to be proportional to a client dynamic analysis's run-time

	Known races	Static analysis unsound		Static analysis sound	Caper	Dyn. alias analysis
hsqldb6	10	13,749	(5)	212,205	1,612	757
lusearch6	0	395	(0)	4,692	302	292
xalan6	17	70,263	(7)	83,488	1,241	581
avrora9	7	1,301	(5)	61,193	19,941	570
luindex9	2	6,015	(0)	10,257	192	193
lusearch9	0	441	(0)	7,303	34	39
sunflow9	2	24,616	(0)	28,587	200	1,086
xalan9	7	13,335	(0)	20,036	1,861	600
pjbb2000	7	12,708	(0)	29,604	11,243	1,679
pjbb2005	1	682	(0)	2,552	984	447

Table 3. Potential data races reported by static analysis, Caper, and dynamic alias analysis. *Known races* are data races reported by a precise detector (see Table 2). For *unsound* static analysis, the number in parentheses is known data races *missing* from the set of reported potential races.

	Static analysis	Caper	Dyn. alias analysis
hsqldb6	97%	44%	71%
lusearch6	73%	57%	52%
xalan6	74%	20%	8%
avrora9	99%	91%	54%
luindex9	33%	4%	< 1%
lusearch9	84%	< 1%	< 1%
sunflow9	65%	14%	53%
xalan9	62%	28%	< 1%
pjbb2000	72%	39%	35%
pjbb2005	97%	26%	8%

Table 4. Percentage of *dynamic memory accesses* that three sound analyses identify as part of a potential data race.

overhead. Table 4 shows the percentage of dynamic accesses that static analysis, Caper, and dynamic alias analysis identify as being part of a potential data race. Each percentage is computed as:

$$\frac{\sum_{s \mid (\exists s' \mid \langle s,s' \rangle \in potentialRaces)} freq(s)}{\sum_s freq(s)}$$

where $potentialRaces$ is the set of potential races identified by the analysis (e.g., $dpPairs$ for Caper), and $freq(s)$ is the dynamic execution frequency of site s.

The table shows that static analysis's high imprecision leads to most executed accesses being potentially racy. Caper improves precision substantially over static analysis for all programs but avrora9. Although dynamic alias analysis usually has the best dynamic precision, Caper provides the best precision for hsqldb6 and sunflow9, due to analysis differences discussed above.

These results suggest that Caper's generated set of potential races can enable significantly reduced costs of dynamic analyses and systems, including race detectors such as RaceChaser. In contrast, static analysis is significantly more imprecise, and dynamic alias analysis is too slow to be run continuously in production.

In summary, the results for Caper suggest that it is an efficient and effective approach that, compared with known alternatives, provides a reasonable tradeoff in *balancing performance and precision* for detecting potential data races.

9. Related Work

Previous sections compared our RaceChaser analysis with closely related work on sampling-based data race detectors including DataCollider and RaceMob's happens-before analysis [28, 37] (Sections 2 and 8.2.1), and Caper with static analysis and dynamic alias analysis. This section compares RaceChaser and Caper with other existing work.

Dynamic escape analysis. Dynamic escape analysis (DEA) identifies when thread-private data becomes shared (Section 5.2). DEA can be implemented based on either "first shared access" [33, 54, 59], reachability from a shared context [21], or variants on these approaches [41, 53].

First-shared-access-based DEA marks each field with its original accessing thread [33, 53, 54, 59].[12] These analyses check at each access whether the current accessing thread is the same as the original accessing thread, thereby detecting the first shared access and marking the field shared. Access-based analysis is *unsound* as a filter for more expensive data race checks. Since no race detection metadata is recorded for accesses to a field before its first shared access, the data race detector cannot determine whether *pre*-sharing accesses race with *post*-sharing accesses on this field.

Reachability-based DEA has the potential to be a sound filter for data race detection, as we show in Section 5.2. However, existing data race detection techniques that make use of reachability-based DEA may miss data races. The *TRaDe* race detection analysis uses sound, reachability-based DEA, but pairs it with an unsound optimization for reprivatization of escaped objects [21]. The *SOS* race detection analysis uses a less precise, but sound, reachability-based DEA as part of a larger *unsound* optimization to detect stationary objects [41]. Reachability-based DEA has been deployed soundly in the implementations of thread-local heaps [24] and software transactional memory [64].

Recent work introduces a field-precise dynamic analysis to precisely track sharing across threads [34], but its overhead is too high for production.

Optimizing data race detection. Wester et al. parallelize happens-before and lockset analyses by using a speculation-based technique called *uniparallelism* [70]. Its performance depends on having extra available cores for speculative execution. Veeraraghavan et al. employ uniparallelism to infer data races, based on different outcomes under deterministic synchronization schedules [67].

Several analyses detect conflicts between regions of code, in order to detect the subset of true data races that may violate *region serializability* in the current execution [9, 17, 25, 43, 47], trading coverage for performance. These techniques either require custom hardware [43, 47], incur substantial slowdowns [9, 25], or use sampling to trade coverage for performance [17].

Custom hardware can accelerate data race detection by adding on-chip memory for tracking vector clocks or locksets, and extending cache coherence to identify conflicts [22, 71, 73].

Increasing coverage. Predictive analysis and model checking enhance race detection coverage without sacrificing precision [29, 35, 48, 65]. These analyses are inherently heavyweight and unsuited for production. Predictive analysis finds data races in an execution *other than* the observed execution [29, 35, 65], but coverage is still limited largely by the observed execution. (*Racageddon* uses a combination of concolic testing and predictive analysis [29].) Model checkers such as *CHESS* [48] can explore many thread interleavings and/or inputs, but they suffer state-space explosion and do not scale well to large programs.

Estimating harmfulness. Prior work tries to infer which data races are most likely to be harmful (e.g., crash the program, hurt performance, or corrupt data) [16, 18, 28, 32, 36, 51, 60]. Several approaches expose errors by exposing rare but allowable behaviors, such as weak memory model behaviors, at racy accesses [16, 18, 32, 51, 60]. Prioritizing races is complementary to our work, which seeks to expose and detect data races. Researchers have argued persuasively that essentially all data races are problematic because languages like Java and C/C++ provide virtually no guarantees for them [1, 12, 13, 45] (Section 1).

[12] Nishiyama's DEA marks an object escaped when a second thread *obtains a direct reference to the object*, which is unsound for race detection [53].

10. Conclusion

Data races are elusive, manifesting unexpectedly in production runs. Sound *and* precise race detection is too expensive for production runs, so we attack soundness and precision separately, introducing complementary approaches that maintain a precise underapproximation and sound overapproximation of true data races—useful inputs both for developers and for analyses and tools. RaceChaser is a novel, precise race detector that adds bounded run-time overhead and outperforms its closest competitor without sacrificing coverage. Caper soundly combines static and dynamic analyses in a novel way to provide significantly better precision than static analysis alone, without the high overhead of more-precise dynamic analyses. Together, these contributions and results demonstrate how to leverage production runs for data race detection effectively.

Acknowledgments

Many thanks to Dan Grossman for valuable discussions and suggestions. We thank Baris Kasikci and the anonymous reviewers for detailed feedback on the text. Thanks to George Candea, Baris Kasikci, Mayur Naik, and Aritra Sengupta for helpful discussions.

References

[1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *ISCA*, pages 234–243, 1991.

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[4] M. Arnold, M. Vechev, and E. Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *OOPSLA*, pages 143–162, 2008.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, pages 81–96, 2009.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM*, 53(2):66–75, 2010.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, pages 72–81, 2008.

[8] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *PLDI*, pages 28–39, 2014.

[9] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[11] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

[12] H.-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[13] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[14] H.-J. Boehm and B. Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.

[15] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[16] J. Burnim, K. Sen, and C. Stergiou. Testing Concurrent Programs on Relaxed Memory Models. In *ISSTA*, pages 122–132, 2011.

[17] Y. Cai, J. Zhang, L. Cao, and J. Liu. A Deployable Sampling Strategy for Data Race Detection. In *FSE*, pages 810–821, 2016.

[18] M. Cao, J. Roemer, A. Sengupta, and M. D. Bond. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *ISMM*, pages 99–110, 2016.

[19] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ASPLOS*, pages 156–164, 2004.

[20] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, 2002.

[21] M. Christiaens and K. De Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *Symposium on Java Virtual Machine Research and Technology Symposium*, pages 15–15, 2001.

[22] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, pages 201–212, 2012.

[23] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*, pages 85–96, 1991.

[24] D. Doligez and X. Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *POPL*, pages 113–123, 1993.

[25] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, pages 467–484, 2012.

[26] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

[27] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[28] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, pages 1–16, 2010.

[29] M. Eslamimehr and J. Palsberg. Race Directed Scheduling of Concurrent Programs. In *PPoPP*, pages 301–314, 2014.

[30] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *SCP*, 71(2):89–109, 2008.

[31] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[32] C. Flanagan and S. N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, pages 244–254, 2010.

[33] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, pages 1–8, 2010.

[34] J. Huang. Scalable Thread Sharing Analysis. In *ICSE*, pages 1097–1108, 2016.

[35] J. Huang, P. O. Meredith, and G. Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI*, pages 337–348, 2014.

[36] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[37] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, pages 406–422, 2013.

[38] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[39] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[40] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[41] D. Li, W. Srisa-an, and M. B. Dwyer. SOS: Saving Time in Dynamic Race Detection with Stationary Analysis. In *OOPSLA*, pages 35–50, 2011.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[43] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[44] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

[45] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[46] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[47] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[48] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[49] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[50] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[51] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31, 2007.

[52] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, pages 89–100, 2007.

[53] H. Nishiyama. Detecting Data Races using Dynamic Escape Analysis based on Read Barrier. In *VMRT*, pages 127–138, 2004.

[54] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, pages 167–178, 2003.

[55] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[56] PCWorld. Nasdaq's facebook glitch came from race conditions, 2012. `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[57] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *CCPE*, 19(3):327–340, 2007.

[58] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, pages 320–331, 2006.

[59] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[60] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[61] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic Race Detection with LLVM Compiler. In *RV*, pages 110–114, 2012.

[62] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[63] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: A Lightweight and Non-invasive Race Detection Tool for Production Applications. In *ICSE*, pages 401–410, 2011.

[64] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, pages 78–88, 2007.

[65] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, pages 387–400, 2012.

[66] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[67] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *SOSP*, pages 369–384, 2011.

[68] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[69] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*, pages 205–214, 2007.

[70] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. Parallelizing Data Race Detection. In *ASPLOS*, pages 27–38, 2013.

[71] B. P. Wood, L. Ceze, and D. Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, pages 671–686, 2014.

[72] X. Xie and J. Xue. Acculock: Accurate and Efficient Detection of Data Races. In *CGO*, pages 201–212, 2011.

[73] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, pages 121–132, 2007.