RaceMob: Crowdsourced Data Race Detection

Baris Kasikci, Cristian Zamfir, and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{baris.kasikci,cristian.zamfir,george.candea}@epfl.ch

Abstract

Some of the worst concurrency problems in multi-threaded systems today are due to data races—these bugs can have messy consequences, and they are hard to diagnose and fix. To avoid the introduction of such bugs, system developers need discipline and good data race detectors; today, even if they have the former, they lack the latter.

We present RaceMob, a new data race detector that has both low overhead and good accuracy. RaceMob starts by detecting potential races statically (hence it has few false negatives), and then dynamically validates whether these are true races (hence has few false positives). It achieves low runtime overhead and a high degree of realism by combining real-user crowdsourcing with a new on-demand dynamic data race validation technique.

We evaluated RaceMob on ten systems, including Apache, SQLite, and Memcached—it detects data races with higher accuracy than state-of-the-art detectors (both static and dynamic), and RaceMob users experience an average runtime overhead of about 2%, which is orders of magnitude less than the overhead of modern dynamic data race detectors. To the best of our knowledge, RaceMob is the first data race detector that can both be used always-on in production and provides good accuracy.

1 Introduction

Data races are at the root of many concurrency-related problems, including atomicity and order violations [29]. Although prevalent in modern software, they rarely affect users: only 5–24% of data races have an observably harmful effect [23, 33, 44]. But when they do have harmful effects, their consequences can be catastrophic. This makes data races the quintessential "corner case": they have effects we really want to avoid, but can only be seen under thread interleavings that have low probability of occurring during testing or normal use—this makes them hard to weed out prior to releasing the software.

Not only are data races omnipresent now, but things are likely to get worse in the future. The new C/C++ standards [19, 20] allow compilers to perform optimizations that, as a side effect, may transform code with data races that look "benign" in the source code into machine code where these data races are seriously harmful [5]. Furthermore, code is running on increasingly more parallel hardware, thus likely to experience more unexpected interleavings. What's worse, practitioners already report that, in real systems, it takes on the order of weeks to diagnose and fix problems whose root cause is a data race bug [16]. This makes such data race bugs very expensive.

Data race detectors play a crucial role in addressing this problem, because they can tell developers where data races lurk in their code, even though developers may choose to not fix them. There exist two broad classes of data race detectors: static and dynamic.

Static data race detectors [10, 44] analyze the program source code without executing it. They reason about multiple program paths at once, and thus typically do not miss data races (i.e., have low rate of false negatives) [36]. Static detectors also run fast and scale to large code bases. The problem is that static data race detectors tend to have many false positives, i.e., produce reports that do not correspond to real data races (e.g., 84% of data races reported by RELAY are not true data races [44]). This can send developers on a wild goose chase, making the use of static detectors potentially frustrating and expensive.

Dynamic data race detectors [18, 40] typically monitor memory accesses and synchronization at runtime, and determine if the observed accesses race with each other. Such detectors can achieve low rates of false positives.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522736

Alas, dynamic detectors miss all the data races that are not seen in the directly observed execution (i.e., they have false negatives), and these can be numerous. The instrumentation required to observe all memory accesses makes dynamic detectors incur high runtime overheads (200× for Intel Thread Checker [18], 30× for Google ThreadSanitizer [40]). As a result, dynamic detectors are not practical for in-production use, rather only during testing—this deprives them of the opportunity to observe real user executions, thus missing data races that only occur in real user environments. Some detectors employ sampling [6, 31] to decrease runtime overhead, but this comes at the cost of further false negatives.

Software developers therefore can get data race detectors with low overhead but high false positive rates *or* detectors with low false positive rates but high overhead and high false negative rates. Either choice gives developers a low-accuracy tool, so data race detectors do not see much use in practice [31].

In this paper, we present RaceMob, a way to combine static and dynamic data race detection to obtain both good accuracy and low runtime overhead. For a given program *P*, RaceMob first uses a static detection phase with few false negatives to find potential data races; in a subsequent dynamic phase, RaceMob crowdsources the validation of these alleged data races to user machines that are anyway running *P*. RaceMob provides developers with a dynamically updated list of data races, split into "confirmed true races", "likely false positives", and "unknown"—developers can use this list to prioritize their debugging attention. To minimize runtime overhead experienced by users of *P*, RaceMob adjusts the complexity of data race validation on-demand to balance accuracy and cost. By crowdsourcing validation, RaceMob amortizes the cost of validation and (unlike traditional testing) gains access to real user executions. RaceMob also helps discovering user-visible failures like crashes or hangs, and therefore helps developers to reason about the consequences of data races. We believe RaceMob is the first data race detector that combines sufficiently low overhead to be always-on with sufficiently good accuracy to improve developer productivity.

This paper makes three contributions: (1) A two-phase static–dynamic approach for detecting data races in real world software in a way that is more accurate than the state of the art; (2) A new algorithm for dynamically validating data races on-demand, which has lower overhead than state-of-the-art dynamic detectors, including those based on sampling; and (3) A crowdsourcing framework that, unlike traditional testing, taps directly into real user executions to detect data races.

We evaluated RaceMob on ten different systems, including Apache, SQLite, and Memcached. It found 106 real data races while incurring an average runtime overhead of 2.32% and a maximum overhead of 4.54%. Three of the data races hang SQLite, four data races races crash Pbzip2, and one data race in Aget causes data corruption. Of all the 841 data race candidates found during the static detection phase, RaceMob labeled 77% as likely false positives. Compared to three state-of-the-art data race detectors [6, 40, 44] and two concurrency testing tools [24, 39], RaceMob has lower overhead and better accuracy than all of them.

2 Background and Challenges

The main challenges faced by data race detectors are runtime overhead (§2.1), false negatives (§2.2), and false positives (§2.3).

2.1 Runtime Overhead

Static data race detectors are not used at runtime, so they do not incur any runtime overhead.

Dynamic data race detectors, however, monitor memory accesses and track the happens-before relationship [25] between them. If two instructions access the same memory location in different threads, at least one of the accesses is a write, and there is no happens-before relationship between the accesses, then a dynamic detector would flag this as a data race.

Such detectors typically need to monitor many memory accesses and synchronization operations, which makes them incur high runtime overhead (as high as 200× in industrial-strength tools like Intel ThreadChecker [18]). The lion's share of instrumentation overhead is due to monitoring memory reads and writes, reported to account for as much as 96% of all monitored operations [13]. Sampling-based data race detectors [6, 31] reduce this overhead but also introduce more false negatives than classic dynamic data race detectors.

Goldilocks [9] uses thread escape analysis [32] to reduce the set of memory accesses that need to be monitored at runtime. A similar approach was proposed earlier by Choi. et al. [7], using a variant of escape analysis. Despite this analysis, the detectors still incur overheads that make them impractical for in-production use.

Our key objective is to have a good data race detector that can be used in an always-on fashion in production. This is why RaceMob uses static analysis to reduce the number of memory accesses that need to be monitored at runtime (§3.1), thereby reducing overhead by up to two orders of magnitude compared to existing sampling-based techniques (§5.3), while also being more accurate.

2.2 False Negatives

Besides runtime overhead, dynamic data race detectors also typically have false negatives. The key reasons are: (1) they can at best detect data races in the executions they witness, which is typically only a tiny subset of a program's possible executions; and (2) while monitoring even this small subset of executions, they may incorrectly infer happens-before relationships that are mere artifacts of the witnessed thread interleaving.

To illustrate point (2), consider Fig. 1. In execution 1, the accesses to the shared variable x are ordered by an accidental happens-before relationship (due to a fortuitous ordering of the acquire/release order of locks) that masks the true data race. Therefore, a precise dynamic detector would not flag this as a data race. However, this program does have a race, which becomes visible under a different schedule. This is shown in execution 2, where there is no happens-before relationship between accesses to x; a precise dynamic detector would have reported a data race only if it witnessed this latter thread schedule.

Hybrid data race detectors [36] combine lockset-based data race detection [38] with happens-before detection to overcome this limitation. However, hybrid data race detectors cannot explore the consequences of such "hidden" data races. They merely infer the presence of potential data races, but also have false positives due to the imprecise lockset analysis.

We wish to minimize the number of false negatives, because every missed race is potentially a serious bug that might compromise security, safety, or other important system properties. In order to minimize false negatives, RaceMob mitigates point (1) above with crowdsourcing (§3.3), and point (2) by exposing hidden data races through schedule steering (§3.2.3).

2.3 False Positives

Dynamic detectors usually do not have false positives. Practical static detectors, however, are notorious for having many false positives (e.g., 84% for RELAY [44]), because they do not reason about the program's full runtime execution context. The problem with getting many false positives is that they overwhelm the developers and make them waste time investigating the race reports [16].

Practical static race detectors have three main sources of false positives. First, they do not accurately infer which program contexts are multithreaded. Second, they typically handle lock/unlock synchronization primitives but not other primitives, such as barriers, semaphores, or wait/notify constructs. Third, static detectors cannot determine accurately whether two memory accesses alias or not. Some static race detectors cope by employing unsound filtering to reduce false positives; while this improves the false positive rate, it still remains high (e.g., RacerX [10] has $37\% - 46\%$ false positive rate). More importantly, such filtering introduces false negatives.

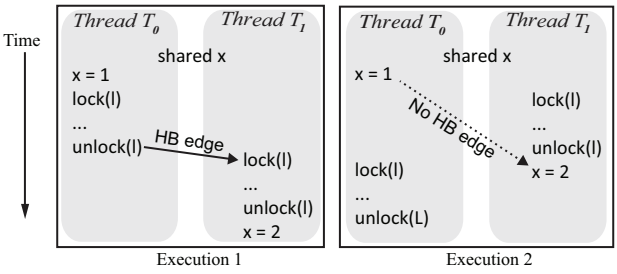

Figure 1: False negatives in happens-before (HB) dynamic race detectors: the race on x is not detected in Execution 1, but it is detected in Execution 2.

3 Design

RaceMob is a crowdsourced, two-phase static–dynamic data race detector. It first statically detects potential data races in a program, then crowdsources the dynamic task of validating these potential data races to users' sites. This validation is done using an on-demand data race detection algorithm. The benefits of crowdsourcing are twofold: first, data race validation occurs in the context of real user executions; second, crowdsourcing amortizes the per-user validation cost. Data race validation confirms true data races, thereby increasing the data race detection coverage[1].

The usage model is presented in Fig. 2. First, developers set up a "hive" service for their program P; this hive can run centralized or distributed. The hive performs static data race detection on P and finds potential data races (§3.1); these go onto P's list of data races maintained by the hive, and initially each one is marked as "Unknown". Then the hive generates an instrumented binary P', which users download ① and use instead of the original P. The instrumentation in P' is commanded by the hive, to activate the validation of specific data races in P ②; different users will typically be validating different, albeit potentially overlapping, sets of data races from P (§3.2). The first phase of validation, called dynamic context inference (§3.2.1), may decide that a particular racing interleaving for data race r is feasible, at which point it informs the hive ③. At this point, the hive instructs all copies of P' that are validating r to advance r to the second validation phase ④. This second phase runs RaceMob's on-demand detection al-

[1] We define data race detection coverage as the ratio of true data races found in a program by a detector to the total number of true data races in that program.

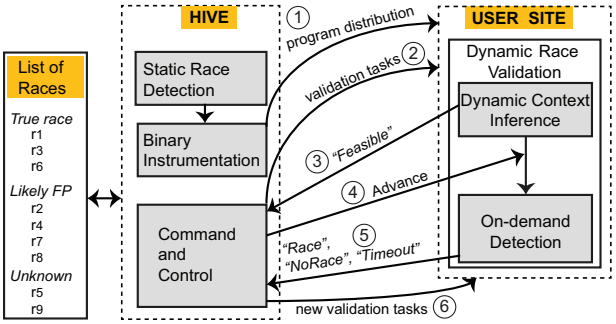

Figure 2: RaceMob's crowdsourced architecture: A static detection phase, run on the hive, is followed by a dynamic validation phase on users' machines.

gorithm (§3.2.2), whose result can be one of Race, No-Race, or Timeout ⑤. As results come in to the hive, it updates the list of data races: if a "Race" result came in from the field for data race r, the hive promotes r from "Unknown" to "True Race"; the other answers are used to decide whether to promote r from "Unknown" to "Likely False Positive" or not (§3.4). For data races with status "Unknown" or "Likely False Positive," the hive redistributes "validation tasks" ⑥ among the available users (§3.3). We now describe each step in further detail.

3.1 Phase I: Static Data Race Detection

RaceMob can use any static data race detector, regardless of whether it is complete or not. We chose RELAY, a lockset-based data race detector [44]. Locksets describe the locks held by a program at any given point in the program. RELAY performs its analysis bottom-up through the program's control flow graph while computing function summaries that summarize which variables are accessed and which locks are held in each function. RELAY then composes these summaries to perform data race detection: it flags a data race whenever it sees at least two accesses to memory locations that are the same or may alias, and at least one of the accesses is a write, and the accesses are not protected by at least one common lock.

RELAY is complete (i.e., does not miss data races) if the program does not have inline assembly and does not use pointer arithmetic. RELAY may become incomplete if configured to perform file-based alias analysis or aggressive filtering, but we disable these options in RaceMob. As suggested in [27], it might be possible to make RELAY complete by integrating program analysis techniques for assembly code [4] and by handling pointer arithmetic [45].

Based on the data race reports from RELAY, RaceMob instruments the suspected-racing memory accesses as well as all synchronization operations in the program. This instrumentation will later be commanded (in production) by RaceMob to perform on-demand data race detection.

The hive activates parts of the instrumentation on-demand when the program runs, in different ways for different users. The activation mechanism aims to validate as many data races as possible by uniformly distributing the validation tasks across the user population.

3.2 Phase II: Dynamic Validation

The hive instructs the instrumented programs for which memory accesses to perform data race validation. The validation task sent by the hive to the instrumented program consists of a data race candidate to validate and one desired order (of two possible) of the racing accesses. We call these possible orders the *primary* and the *alternate*, borrowing terminology from our earlier work [33].

The dynamic data race validation phase has three stages: dynamic context inference (§3.2.1), on-demand data race detection (§3.2.2), and schedule steering (§3.2.3). Instrumentation for each stage is present in all the programs, however stages 2 and 3 are toggled on/off separately from stage 1, which is always on. Next, we explain each stage in detail.

3.2.1 Dynamic Context Inference

Dynamic context inference (DCI) is a lightweight analysis that partly compensates for the inaccuracies of the static data race detection phase. RaceMob performs DCI to figure out whether the statically detected data races can occur in a dynamic program execution context.

DCI validates two assumptions made by the static data race detector about a race candidate. First, the static detector's abstract analysis hypothesizes aliasing as the basis for some of its race candidates, and DCI looks for concrete instances that can validate this hypothesis. Second, the static detector hypothesizes that racing accesses are made by different threads, and DCI aims to validate this as well. Once these two hypotheses are confirmed, the user site communicates this to the hive, and the hive promotes the race candidate to the next phase. Without a confirmation from DCI, the race remains in the "Unknown" state.

The motivation for DCI comes from our observation that the majority of the potential data races detected by static data race detection (53% in our evaluation) are false positives due to only alias analysis inaccuracies and the inability of static race detection to infer multithreaded program contexts.

For every potential data race r with racing instructions r_1 and r_2, made by threads T_1 and T_2, respectively, DCI

determines whether the potentially racing memory accesses to addresses a_1 and a_2 made by r_1 and r_2, respectively, may alias with each other (i.e., $a_1 = a_2$), and whether these accesses are indeed made by different threads (i.e., $T_1 \neq T_2$). To do this, DCI keeps track of the address that each potentially racing instruction accesses, along with the accessing thread's ID at runtime. Then, the instrumentation checks to see if *at least one* pair of accesses is executed. If yes, the instrumented program notifies the hive, which promotes r to the next stages of validation (on-demand data race detection and schedule steering) on all user machines where r is being watched. If no access is executed by any instrumented instance of the program, DCI continues watching r's potentially racing memory accesses until further notice.

DCI has negligible runtime overhead (0.01%) on top of the binary instrumentation overhead (0.77%); therefore, it is feasible to have DCI always-on. DCI's memory footprint is small: it requires maintaining 12 bytes of information per potential racing instruction (8 bytes for the address, 4 bytes for the thread ID). DCI is sound because, for every access pair that it reports as being made from different threads and to the same address, DCI has clear concrete evidence from an actual execution. DCI is of course not guaranteed to be complete.

3.2.2 On-Demand Data Race Detection

In this section, we explain how on-demand data race detection works; for clarity, we restrict the discussion to a single potential data race.

On-demand race detection starts tracking happens-before relationships once the first potentially racing access is made, and it stops tracking once a happens-before relationship is established between the first accessing thread and all the other threads in the program (in which case a "NoRace" result is sent to the hive). Tracking also stops if the second access is made before such a happens-before relationship is found (in which case a "Race" result is sent to the hive).

Intuitively, RaceMob tracks a minimal number of accesses and synchronization operations. RaceMob needs to track both racing accesses to validate a potential data race. However, RaceMob does not need to track any memory accesses other than the target racing accesses, because any other access is irrelevant to this data race.

Sampling-based data race detection (e.g., PACER [6]) adopts a similar approach to on-demand race detection by tracking synchronization operations whenever sampling is enabled. The drawback of PACER's approach is that it may start to track synchronization operations too soon, even if the program is not about to execute a racing access. RaceMob avoids this by turning on tracking synchronization operations on-demand, when an access

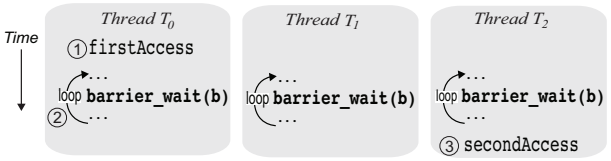

Figure 3: Minimal monitoring in DCI: For this example, DCI stops tracking synchronization operations as soon as each thread goes once through the barrier.

reported by the static race detection phase is executed.

RaceMob tracks synchronization operations—and thus, happens-before relationships—using an efficient, dynamic, vector-clock algorithm similar to DJIT$^+$ [37]. We maintain vector clocks for each thread and synchronization operation, and the clocks are partially ordered with respect to each other.

We illustrate in Fig. 3 how the on-demand race detection stops tracking synchronization operations, using a simple example derived from the `fmm` program [41]: *firstAccess* executes in the beginning of the program in T_0 ①, and the program goes through a few thousand iterations of synchronization-intensive code ②. Finally, T_2 executes *secondAccess* ③. It is sufficient to keep track of the vector clocks of all threads only up until the first time they go through the `barrier_wait` statement, as this establishes a happens-before relationship between *firstAccess* in T_0 and any subsequent access in T_1 and T_2. Therefore, on-demand data race detection stops keeping track of the vector clocks of threads T_0, T_1, and T_2 after they each go through `barrier_wait` once.

3.2.3 Schedule Steering

The schedule steering phase further improves RaceMob's data race detection coverage by exploring both the primary and the alternate executions of potentially racing accesses. This has the benefit of detecting races that may be hidden by accidental happens-before relationships (as discussed in §2.2 and Fig. 1).

Schedule steering tries to enforce the order of the racing memory accesses provided by the hive, i.e., either the primary or the alternate. Whenever the intended order is about to be violated (i.e., the undesired order is about to occur), RaceMob pauses the thread that is about to make the access, by using a wait operation with a timeout τ, to enforce the desired order. Every time the hive receives a "Timeout" from a user, it increments τ for that user (up to a maximum value), to more aggressively steer it toward the desired order, as described in the next section.

Prior work [33, 39, 24] used techniques similar to schedule steering to detect whether a *known* data race can cause a failure or not. RaceMob, however, uses schedule

steering to increase the likelihood of encountering a suspected race and to improve data race detection coverage.

Our evaluation shows that schedule steering helps RaceMob to find two data races (one in Memcached and the other one in Pfscan) that would otherwise be missed. It also helps RaceMob uncover failures (e.g., data corruption, hangs, crashes) that occur as a result of data races, thereby enabling developers to reason about the consequences of data races and fix the important ones early, before they affect more users. However, users who do not wish to help in determining the consequences of data races can easily turn off schedule steering. We discuss the trade-offs involved in schedule steering in §6.

3.3 Crowdsourcing the Validation

Crowdsourcing is defined as the practice of obtaining needed services, ideas, or content by soliciting contributions from a large group of people and especially from the online community. RaceMob gathers validation results from a large base of users and merges them to come up with verdicts for potential data races.

RaceMob's main motivation for crowdsourcing data race detection is to access real user executions. This enables RaceMob, for instance, to detect the important but often overlooked class of input-dependent data races [24], i.e., races that occur only when a program is run with a particular input. RaceMob found two such races in Aget, and we detail them in §5.1. Crowdsourcing also enables RaceMob to leverage many executions to establish statistical confidence in the detection verdict. We also believe crowdsourcing is more applicable today than ever: collectively, software users have overwhelmingly more hardware than any single software development organization, so leveraging end-users for race detection is particularly advantageous. Furthermore, such distribution helps reduce the per-user overhead to barely noticeable levels.

Validation is distributed across a population of users, with each user receiving only a small number of races to validate. The hive distributes validation tasks, which contain the locations in the program of two memory accesses suspected to be racing, along with a particular order of these accesses. Completing the validation task consists of confirming, in the end-user's instance of the program, whether the indicated ordering of the memory accesses is possible. If a user site receives more than one race to validate, it will first validate the race whose racing instruction is first reached during execution.

There exists a wide variety of possible assignment policies that can be implemented in RaceMob. By default, if there are more users than races, RaceMob initially randomly assigns a single race to each user for validation. Assigned validation tasks that fail to complete within a time bound are then distributed to additional users as well, in order to increase the probability of completing them. Such multiple assignment could be done from the very beginning, in order to reach a verdict sooner. The number of users asked to validate a race r could be based, for example, on the expected severity of r, as inferred based on heuristics or the static analysis phase. Conversely, in the unlikely case that there are more data races to validate than users, the default policy is to initially distribute a single validation task to each user, thereby leaving a subset of the races unassigned. As users complete their validation tasks, RaceMob assigns new tasks from among the unassigned races. Once a data race is confirmed as a true race, it is removed from the list of data races being validated, for all users.

During schedule steering, whenever a race candidate is "stubborn" and does not exercise the desired order despite the wait introduced by the instrumentation, a "Timeout" is sent to the hive. The hive then instructs the site to increase the timeout τ, to more aggressively favor the alternate order; every subsequent timeout triggers a new "Timeout" response to the hive and a further increase in τ. Once τ reaches a configured upper bound, the hive instructs the site to abandon the validation task. At this point, or even in parallel with increasing τ, the hive could assign the same task to additional users.

There are two important steps in achieving low overhead at user sites. First, the timeout τ must be kept low. For example, to preserve the low latency of interactive applications, RaceMob uses an upper bound $\tau \leq 200$ ms; for I/O bound applications, higher timeouts can be configured. Second, the instrumentation at the user site disables schedule steering for a given race after a first steering attempt for a given race, regardless of whether it succeeded or not; this is particularly useful when the racing accesses are in a tight loop. Steering is resumed when a new value of τ comes in from the hive. It is certainly possible that a later dynamic instance of the potentially racing instruction might be able to exercise the desired order, had steering not been disabled; nevertheless, in our evaluation we found that RaceMob achieves higher accuracy than state-of-the art data race detectors.

RaceMob monitors each user's validation workload and aims to balance the global load across the user population. Rebalancing does not require users to download a new version of the program, but rather the hive simply toggles validation on/off at the relevant user sites. In other words, each instance of the instrumented program P' is capable of validating, on demand, any of the races found during the static phase—the hive instructs it which race(s) is/are of interest to that instance of P'.

If an instance of P' spends more time doing data race validation than the overall average, then the hive redistributes some of that instance's validation tasks to other

instances. Additionally, RaceMob reshuffles tasks whenever one program experiences more timeouts than the average. In this way, we reduce the number of outliers, in terms of runtime overhead, during the dynamic phase.

Crowdsourcing offers RaceMob the opportunity to tap into a large number of executions, which makes it possible to only perform a small amount of monitoring per user site without harming the precision of detection. This in turn reduces RaceMob's runtime overhead, making it more palatable to users and easier to adopt.

3.4 Reaching a Verdict

The hive receives from the instrumented program instances three possible results: Race, NoRace, or Timeout. After aggregating these over all users, the hive reaches one of three verdicts: "True Race", "Likely False Positive", or "Unknown" (Fig. 4). RaceMob does not currently quantify its statistical confidence in these verdicts, but this could easily be done.

True Race RaceMob decides a race candidate is a true race whenever both the primary and the alternate orders are executed at a user site, or when either of the orders is executed with no intervening happens-before relationship between the corresponding memory accesses. Among the true races, some can be specification-violating races in the sense of [24] (e.g., that cause crash or deadlock). In the case of a crash, the RaceMob instrumentation catches the SIGSEGV signal and submits a crash report to the hive. In the case of an unhandled SIGINT (e.g., the user pressed Ctrl-C), RaceMob prompts the user with a dialog asking whether the program has failed to meet expectations. If yes, the hive is informed that the enforced schedule leads to a specification violation. Of course, users who find this kind of "consequence reporting" too intrusive can disable schedule steering altogether.

Likely False Positive RaceMob concludes that a potential race is likely a false positive if at least one user site reported a NoRace result to the hive (i.e., on-demand race detection discovered a happens-before relationship between the accesses in the primary or alternate). RaceMob cannot provide a definitive verdict on whether the race is a false positive or not, because there might exist some other execution in which the purported false positive proves to be a real race (e.g., due to an unobserved input dependency). The "Likely False Positive" verdict, especially if augmented with the number of corresponding NoRace results received at the hive, can help developers decide whether to prioritize for fixing this particular race over others. RaceMob continues validation for "Likely False Positive" data races for as long as the developers wishes.

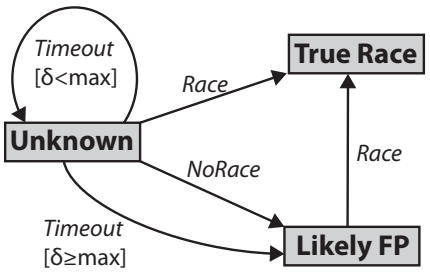

Figure 4: The state machine used by the hive to reach verdicts based on reports from program instances. Transition edges are labeled with validation results that arrive from instrumented program instances; states are labeled with RaceMob's verdict.

Unknown As long as the hive receives no results from the validation of a potential race r, the hive keeps the status of the race "Unknown". Similarly, if none of the program instances report that they reached the maximum timeout value, r's status remains "Unknown". However, if at least one instance reaches the maximum timeout value for r, the corresponding report is promoted to "Likely False Positive".

The "True Race" verdict is definite: RaceMob has proof of the race occurring in a real execution of the program. The "Likely False Positive" verdict is probabilistic: the more NoRace or Timeout reports are received by the hive as a result of distinct executions, the higher the probability that a race report is indeed a false positive, even though there is no precise probability value that RaceMob assigns to this outcome.

4 Implementation

We implemented RaceMob in 4,147 lines of C++ (instrumentation code) and 2,850 lines of Python (hive and user-side daemon). To reduce contention to a minimum, the instrumentation uses thread-local data structures, synchronization operations partitioned by race report, and C++11 weak atomic load/store operations that rely on relaxed memory ordering. In this way, we avoid introducing undue contention in the monitored application; together with crowdsourcing, this is key to keeping RaceMob's runtime overhead low and to scaling to a large number of application threads.

RaceMob can use any data race detector that outputs data race candidates; preferably it should be complete (i.e., not miss data races). We use RELAY, which analyzes programs that are turned into CIL, an intermediate

language for C programs [34]. The instrumentation engine at the hive is based on LLVM [26]. We wrote a 500-LOC plugin that converts RELAY reports to the format required by our instrumentation engine.

The instrumentation engine is an LLVM static analysis pass. It avoids instrumenting empty loop bodies that have a data race on a variable in the loop condition (e.g., of the form `while(notDone){}`). These loops occur often in ad-hoc synchronization [46]. Not instrumenting such loops avoids excessive overhead that results from running the instrumentation frequently. When such loops involve a data race candidate, they are reported by the hive directly to developers. We encountered this situation in two of the programs we evaluated, and both cases were true data races (thus validating prior work that advocates against ad-hoc synchronization [46]), so this optimization did not effect RaceMob's accuracy.

Whereas Fig. 2 indicates three possible results from user sites (Race, NoRace, and Timeout), our prototype also implements a fourth one (NotSeen), to indicate that a user site has not witnessed the race it was expected to monitor. Technically, NotSeen can be inferred by the hive from the absence of any other results. However, for efficiency purposes, we have a hook at the exit of `main`, as well as in the signal handlers, that send a NotSeen message to the hive whenever the program terminates without having made progress on the validation task.

Our prototype can be obtained from the RaceMob website (http://dslab.epfl.ch/proj/racemob).

5 Evaluation

In this section, we address the following questions about RaceMob: Can it effectively detect true races in real code (§5.1)? Is it efficient (§5.2)? How does it compare to state-of-the-art data race detectors (§5.3) and interleaving-based concurrency testing tools (§5.4)? Finally, how does RaceMob scale with the number of threads (§5.5)?

We evaluated RaceMob using a mix of server, desktop and scientific software: Apache httpd is a Web server that serves around 65% of the Web [17]—we used the `mpm-worker` module of Apache to operate it in multi-threaded server mode and detected races in this specific module. SQLite [42] is an embedded database used in Firefox, iOS, Chrome, and Android, and has 100% branch coverage with developer's tests. Memcached [12] is a distributed memory-object caching system, used by Internet services like Twitter, Flickr, and YouTube. Knot [43] is a web server. Pbzip2 [14] is a parallel implementation of the popular `bzip2` file compressor. Pfscan [11] is a parallel file scanning tool that provides the combined functionality of `find`, `xargs`, and `fgrep` in a parallel way. Aget is a parallel variant of `wget`. Fmm, Ocean, and Barnes are applications from the SPLASH suite [41]. Fmm and Barnes simulate interactions of bodies, and Ocean simulates ocean movements.

Our evaluation results are obtained primarily using a test environment simulating a crowdsourced setting, and we also have a small scale, real deployment of RaceMob on our laptops. For the experiments, we use a mix of workloads derived from actual program runs, test suites, and test cases devised by us and other researchers [48]. We configured the hive to assign a single dynamic validation task per user at a time. Altogether, we have execution information from $1,754$ simulated user sites. Our test bed consists of a 2.3 GHz 48-core AMD Opteron 6176 machine with 512 GB of RAM running Ubuntu Linux 11.04 and a 2 GHz 8-core Intel Xeon E5405 machine with 20 GB of RAM running Ubuntu Linux 11.10. The hive is deployed on the 8-core machine, and the simulated users on both machines. The real deployment uses ThinkPad laptops with Intel 2620M processors and 8 GB of RAM, running Ubuntu Linux 12.04.

We used C programs in our evaluation because RELAY operates on CIL, which does not support C++ code. Pbzip2 is a C++ program, but we converted it to C by replacing references to STL `vector` with an array-based implementation. We also replaced calls to `new/delete` with `malloc/free`.

5.1 Effectiveness of Data Race Detection

To investigate whether RaceMob is an effective way to detect data races, we look at whether RaceMob can detect true data races, and whether its false positive and false negative rates are sufficiently low.

RaceMob's data race detection results are centralized in Table 1. RaceMob detected a total of 106 data races in ten programs. Four races in Pbzip2 caused the program to crash, three races in SQLite caused the program to hang, and one race in Aget caused a data corruption (that we confirmed manually). The other races did not lead to any observable failure. We manually confirmed that the "True Race" verdicts are correct, and that RaceMob has no false positives in our experiments.

The "Likely FP" row represents the races that RaceMob identified as likely false positives: (1) *Not aliasing* are reports with accesses that do not alias to the same memory location at runtime; (2) *Context* are reports whose accesses are only made by a single thread at runtime; (3) *Synchronization* are reports for which, the accesses are synchronized, an artifact that the static detector missed. The first two sources of likely false positives (53% of all static reports) are identified using DCI, whereas the last source (24% of all static reports) is iden-

Program		Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
Size (LOC)		138,456	113,326	19,397	9,126	7,580	6,551	3,521	3,586	2,053	2,033
Race candidates		118	88	7	176	166	115	65	65	24	17
True Race	Causes hang	0	3	0	0	0	0	0	0	0	0
	Causes crash	0	0	0	0	0	0	3	0	0	0
	Both orders	0	0	1	5	10	0	2	0	0	0
	Single order	8	0	0	53	6	3	4	2	4	2
Likely FP	Not aliasing	10	31	0	33	65	13	0	18	2	0
	Context	61	10	2	61	28	42	21	28	10	4
	Synchronization	1	37	3	10	49	47	34	13	7	11
Unknown		38	7	1	14	8	10	1	4	1	0

Table 1: Data race detection with RaceMob. The static phase reports *Race candidates* (row 2). The dynamic phase reports verdicts (rows 3-10). *Causes hang* and *Causes crash* are races that caused the program to hang or crash. *Single order* are true races for which either the primary or the alternate executed (but not both) with no intervening synchronization; *Both orders* are races for which both executed without intervening synchronization.

tified using on demand race detection. In total, 77% of all statically detected races are likely false positives.

As we discussed in §3.4, RaceMob's false negative rate is determined by its static data race detector. We manually verified that none of RELAY's sources of false negatives (i.e., inline assembly and pointer arithmetic) are present in the programs in our evaluation. Furthermore, Chimera [27], a deterministic record/replay system, relies on RELAY; for deterministic record/replay to work, all data races must be detected; in Chimera's evaluation (which included Apache, Pbzip2, Knot, Ocean, Pfscan, Aget), RELAY did not have any false negatives [27]. We therefore cautiously conclude that RaceMob's static phase had no false negatives in our evaluation. However, this does not exclude the possibility that for other programs there do exist false negatives.

For all the programs, we initially set the timeout for schedule steering to $\tau = 1$ ms. As timeouts fired during validation, the hive increased the timeout 50 ms at a time, up to a maximum of 200 ms. Developers may choose to adapt this basic scheme depending on the characteristics of their programs. For instance, the timeout could be increased multiplicatively instead of linearly.

In principle, false negatives may also arise from τ being too low or from there being insufficient executions to prove a true race. We increased τ in our experiments by $4\times$, to check if this would alter our results, and the final verdicts were the same. After manually examining races that were not encountered during dynamic validation, we found that they were either in functions that are never called but are nonetheless linked to the programs, or they are not encountered at runtime due to the workloads used in the evaluation.

5.2 Efficiency

The more efficient a detector is, the less runtime overhead it introduces, i.e., the less it slows down a user's application (as a percentage of uninstrumented execution). The static detection phase is offline, and it took less than 3 minutes for all programs, except Apache and SQLite, for which it took less than 1 hour. Therefore, in this section, we focus on the dynamic phase.

Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
1.74	1.60	0.10	4.54	2.98	2.05	2.90	1.27	3.00	3.03

Table 2: Runtime overhead of race detection as a percentage of uninstrumented execution. Average overhead is 2.32%, and maximum overhead is 4.54%.

Table 2 shows that runtime overhead of RaceMob is typically less than 3%. The static analysis used to remove instrumentation from empty loop bodies reduced our worst case overhead from 25% to 4.54%. The highest runtime overhead is 4.54%, in the case of Fmm, a memory-intensive application that performs repetitive computations, which gives the instrumentation more opportunity to introduce overhead. Our results suggest that there is no correlation between the number of race candidates (row 2 in Table 1) and the runtime overhead (Table 2)—overhead is mostly determined by the frequency of execution of the instrumentation code.

The overhead introduced by RaceMob is due to the instrumentation plus the overhead introduced by validation (DCI, on-demand detection, and schedule steering). Fig. 5 shows the breakdown of overhead for our ten tar-

Program	Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
RaceMob	8	3	1	58	16	3	9	2	4	2
TSAN	8	3	0	58	16	3	9	2	2	1
RELAY	118	88	7	176	166	115	65	157	256	17

Table 3: Race detection results with RaceMob, ThreadSanitizer (TSAN), and RELAY. Each cell shows the number of reported races. The data races reported by RaceMob and TSAN are all true data races. The only true data races among the ones detected by RELAY are the ones in the row "RaceMob". To the best of our knowledge, two of the data races that cause a hang in SQLite were not previously reported.

get programs. We find that the runtime overhead without detection is below 1% for all cases, except the memory-intensive Fmm application, for which it is 2.51%. We conclude that, in the common case when a program is instrumented by RaceMob but no detection is performed, the runtime overhead is negligible; this property is what makes RaceMob suitable for always-on operation.

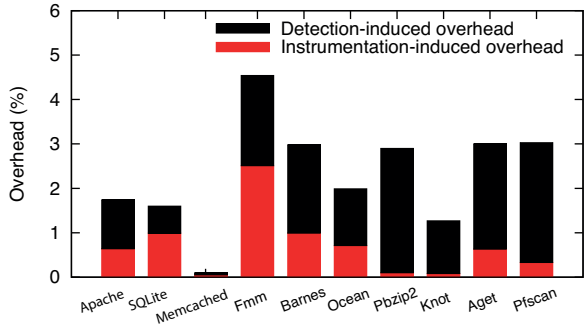

Figure 5: Breakdown of average overhead into instrumentation-induced overhead and detection-induced overhead.

The dominant component of the overhead of race detection (the black portion of the bars in Fig. 5) is due to dynamic data race validation. The effect of DCI is negligible: it is below 0.1% for all cases; thus, we don't show it in Fig. 5. Therefore, it is feasible to leave DCI on for all executions. This can help RaceMob to promote a race from "Likely FP" to "True Race" with low overhead.

If RaceMob assigns more than one validation task at a time per user, the aggregate overhead that a user experiences will increase. In such a scenario, the user site would pick a validation candidate at runtime depending on which potentially racing access is executed. This scheme introduces a lookup overhead to determine at runtime which racing access is executed, however, it would not affect the per-race overhead, because of RaceMob's on-demand race detection algorithm.

5.3 Comparison to Other Detectors

In this section, we compare RaceMob to state-of-the art dynamic, static, and sampling-based race detectors.

We compare RaceMob to the RELAY static data race detector [44] and to ThreadSanitizer [40] (TSAN), an open-source dynamic race detector developed by Google. We also compare RaceMob to PACER [6], a sampling-based race detector. Our comparison is in terms of detection results and runtime overhead. We do not compare to LiteRace, which is another sampling-based data race detector, because LiteRace has higher overhead and lower data race detection coverage than PACER [31]. The detection results are shown in Table 3.

5.3.1 Comparative Accuracy

We first compared RaceMob to TSAN by detecting races for all the test cases that were available to us, except for the program executions from the real deployment of RaceMob, because we do not record real user executions. RaceMob detected 4 extra races relative to TSAN: For Memcached and Pfscan, RaceMob detected, with the help of schedule steering, 2 races missed by TSAN. RaceMob also detected 2 input-dependent races in Aget that were missed by TSAN (of which one causes Aget to corrupt data), because RaceMob had access to executions from the real deployment, which were not accessible to TSAN. These races required the user to manually abort and restart Aget. For 3 races in Pbzip2, RaceMob triggered a particular interleaving that caused the program to crash as a result of schedule steering, which did not happen in the case of TSAN. Furthermore, we have not observed any crash during detection with TSAN; this shows that, without schedule steering, the consequences of a detected race may remain unknown.

Note that we give TSAN the benefit of access to all executions that RaceMob has access to (except the executions from the real users). This is probably overly generous, because in reality, dynamic race detection is not crowdsourced, so one would run TSAN on fewer executions and obtain lower data race detection coverage than shown here. We did not use TSAN's hybrid data race detection algorithm, because it is known to report false positives and therefore lower the accuracy of data race detection.

RELAY typically reports at least an order of magnitude more races than the real races reported by RaceMob, with

Program	Aggregate overhead with RaceMob [# of race candidates × # of users] in %	TSAN user-perceived overhead in %
Apache	339.30	25,207.79
SQLite	281.60	1,428.57
Memcached	2.20	3,102.32
Fmm	1,598.08	47,888.07
Barnes	989.36	30,640.00
Ocean	360.70	3,069.39
Pbzip2	377.00	3,001.00
Knot	165.10	751.47
Aget	144.00	184.22
Pfscan	103.20	13,402.15

Table 4: RaceMob aggregate overhead vs. TSAN's average overhead, relative to uninstrumented execution. RaceMob's aggregate overhead is across all the executions for all users. For TSAN, we report the average overhead of executing all the available test cases.

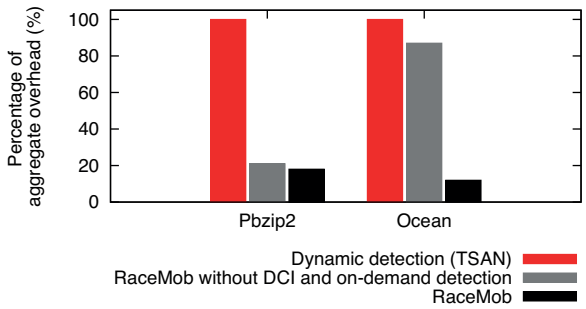

Figure 6: Contribution of each technique to lowering the aggregate overhead of RaceMob. Dynamic detection represents detection with TSAN. RaceMob without DCI and on-demand detection just uses static data race detection to prune the number of accesses to monitor.

no indication of whether they are true races or not. Consequently, the developers would not have information on how to prioritize their bug fixing. This would in turn impact the users, because it might take longer to remove the data races with severe consequences. The benefit of tolerating a 2.32% average detection overhead with RaceMob is that race detection results are more detailed and helpful. To achieve a similar effect as RaceMob, static data race detectors use unsound heuristics to prune some race reports, and thus introduce false negatives.

5.3.2 Comparative Overhead

RELAY's static data race detection is offline, and the longest detection we measured was below 1 hour.

We compared the overheads of dynamic race detection in RaceMob and TSAN. We chose TSAN because it is freely available, actively maintained, and works for C programs. The results are shown in Table 4. The average overhead of TSAN ranged from almost 49× for Fmm to 1.84× for Aget. The average overhead of RaceMob per user is about three orders of magnitude less than that of TSAN for all three programs.

The aggregate overhead of RaceMob represents the sum of all the overheads of all the executions at all the user sites. It represents RaceMob's overall overhead for detecting the data races in row 2 of Table 3. We compare RaceMob's aggregate overhead to TSAN's overhead because these overheads represent what both tools incur for all the races they detect. The aggregate overhead of RaceMob is an order of magnitude less than the overhead of TSAN. This demonstrates that mere crowdsourcing of TSAN would not be enough to reduce its overhead (it would still be one order of magnitude away from RaceMob), and so the other techniques proposed in RaceMob are necessary too.

In particular, there are two other factors that contribute to lower overhead: the static race detection phase and the lightweight dynamic validation phase. The contribution of each such phase depends on whether the application for which RaceMob performs race detection is synchronization-intensive or not. To show the benefit of each phase, we picked Ocean (synchronization-intensive) and Pbzip2 (uses less synchronization), and measured the contribution of each phase.

The results are shown in Fig. 6. This graph shows how the overhead of full dynamic detection reduces with each phase. The contribution of static race detection is more significant for Pbzip2 in comparison to Ocean. This is because, for Pbzip2, narrowing down the set of accesses to be monitored has a good enough contribution. On the other hand, Ocean benefits more from DCI and on-demand race detection, because static data race detection is inaccurate in this case (and is mitigated by DCI), and Ocean employs heavy synchronization (mitigated by on-demand data race detection). Thus, we conclude that both the static race detection phase and DCI followed by on-demand race detection are essential to lowering the overhead of aggregate race detection in the general case.

We also compared the runtime overhead with PACER, a sampling-based data race detector. We do not have access to a PACER implementation for C/C++ programs; therefore, we modified RaceMob to operate like PACER. We allow PACER to have access to the static race detection results from RELAY, and we assumed PACER starts sampling whenever a potential racing access is performed (as in RaceMob) rather than at a random time. We refer to our version of PACER as PACER-SA.

PACER-SA's runtime overhead is an order of

magnitude larger than that of RaceMob for non-synchronization-intensive programs: 21.56% on average for PACER-SA vs. 2.32% for RaceMob. RaceMob has lower overhead mainly because it performs race detection selectively: it does not perform on-demand race detection for every potential race detected statically, rather it only does so after DCI has proven that the relevant accesses can indeed alias and that they indeed can occur in a multithreaded context. Table 1 shows that DCI excludes on this basis more than half the race candidates from further analysis.

For synchronization-intensive programs, like Fmm, Ocean and Barnes, PACER-SA's overhead can become up to two orders of magnitude higher than that of RaceMob. This is due to the combined effect of DCI and on-demand race detection. The latter factor is more prominent for synchronization-intensive applications. To illustrate this, we picked Fmm and used RaceMob and PACER-SA to detect races. For typical executions of 200 msec, where we ran Fmm with its default workload, Fmm performed around 15,000 synchronization operations, which incur a 200% runtime overhead with PACER-SA compared to 4.54% with RaceMob.

We conclude that, even if PACER-SA's performance might be considered suitable for production use for non-synchronization-intensive programs, it is prohibitively high in the case of synchronization-intensive programs. This is despite giving the benefit of a static race detection phase to vanilla PACER. PACER could have lower overhead than RaceMob if it stopped sampling soon after having started and before even detecting a data race, but it would of course also detect fewer data races.

This section showed that RaceMob detects more true races than state-of-the art detectors while not introducing additional false negatives relative to what the static race detectors already do. It also showed that RaceMob's runtime overhead is lower than state-of-the-art detectors.

5.4 Concurrency Testing Tools

A concurrency testing tool can be viewed as a type of race detector, and vice versa. In this vein, one could imagine using RaceMob for testing, by using schedule steering (§3.2.3) to explore races that may otherwise be hard to witness and that could lead to failures. As a simple test, we ran SQLite with the test cases used in our evaluation 10,000 times and never encountered any hang when not instrumented. When running it under RaceMob, we encountered 3 hangs within 176 executions. Similarly, we ran the Pbzip2 test cases 10,000 times and never encountered a crash, but RaceMob caused the occurrence of 4 crashes within 130 executions. This suggests that RaceMob could also be used as a testing tool to quickly identify and prioritize data race bugs.

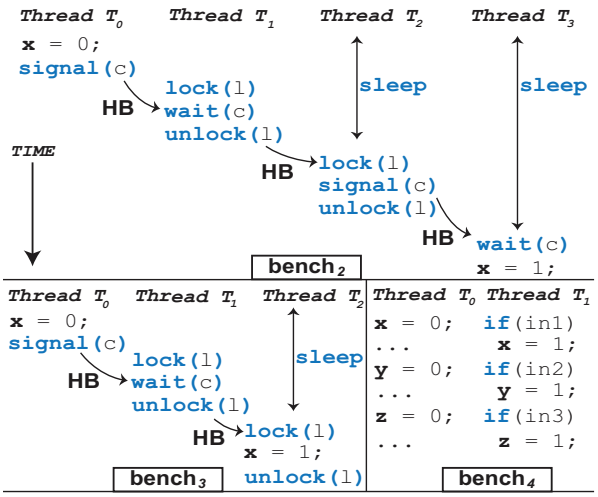

Figure 7: Concurrency testing benchmarks: bench$_1$ is shown in Fig. 1, thus not repeated here. In bench$_2$, the accesses to x in T_0 and T_2 can race, but the long sleep in T_2 and T_3 causes the signal-wait and lock-unlock pairs to induce a happens-before edge between T_0 and T_3. bench$_3$ has a similar situation to bench$_2$. In bench$_4$, the accesses to variables x, y, z from T_0 and T_1 are racing if the input is either in_1, in_2, or in_3.

Existing concurrency testing tools perform an analysis similar to schedule steering to detect and explore races. In the rest of this section we compare RaceMob to two such state-of-the-art tools: RaceFuzzer [39] and Portend [24]. These tools were not intended for use in production, and thus have high overheads (up to 200× for RaceFuzzer and up to 5,000× for Portend), so we do not compare on overhead, but focus instead on comparing their respective data race detection coverage.

RaceFuzzer works in two stages: First, it uses imprecise hybrid race detection [36] to detect potential races in a program and instrument them. Second, it uses a randomized analysis to determine whether these potential races are actual races. Portend uses precise happens-before dynamic race detection and explores a detected race's consequences along multiple paths and schedules.

To compare data race detection coverage, we use benchmarks bench$_1$, bench$_2$, bench$_3$ (taken from Google TSAN) and bench$_4$ (taken from the Portend paper [24]). The bench$_4$ benchmark has three races that only manifest under specific inputs in_1, in_2, and in_3. Simplified versions of the benchmarks are shown in Fig. 7 and Fig. 1.

The RaceFuzzer implementation is not available, so we simulate it: we use TSAN in imprecise hybrid mode, as done in RaceFuzzer, and then implement RaceFuzzer's random scheduler. The results appear in Table 5. For bench$_1$, bench$_2$, and bench$_3$, RaceFuzzer performs as well as RaceMob in terms of data race detection

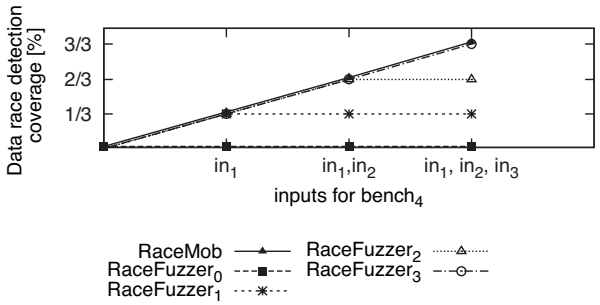

Figure 8: Data race detection coverage for RaceMob vs. RaceFuzzer. To do as well as RaceMob, RaceFuzzer must have a priori access to all test cases (the RaceFuzzer$_3$ curve).

coverage. For bench$_4$, RaceFuzzer's data race detection coverage varies between $0/3 - 3/3$.

Tool	bench$_1$	bench$_2$	bench$_3$	bench$_4$
RaceMob	1 / 1	1 / 1	1 / 1	3 / 3
RaceFuzzer	1 / 1	1 / 1	1 / 1	0 – 3 / 3
Portend	0 / 1	0 / 1	0 / 1	3 / 3

Table 5: RaceMob vs. concurrency testing tools: Ratio of races detected in each benchmark to the total number of races in that benchmark.

To understand this variation, we run the following experiment: we assume that initially neither tool has access to any test case with input in_1, in_2, or in_3. Thus, RaceFuzzer cannot detect any race, so it cannot instrument the racing accesses, and generates an instrumented version of bench$_4$ we call RaceFuzzer$_0$. RaceMob, however, detects all three potential races in bench$_4$, thanks to static race detection, and instruments bench$_4$ at the potentially racing accesses. If we allow RaceFuzzer to see a test with input in_1, then it generates a version of bench$_4$ we call RaceFuzzer$_1$; if we allow it to see both a test with input in_1 and in_2, then it generates RaceFuzzer$_2$. RaceFuzzer$_3$ corresponds to having seen all three inputs.

We run both RaceFuzzer's and RaceMob's versions of the instrumented benchmark and plot data race detection coverage in Fig. 8. When run on random inputs different from in_1, in_2, and in_3, neither tool finds any race (0/3), as expected. When given input in_1, RaceMob finds the race, RaceFuzzer$_0$ doesn't, but RaceFuzzer$_1$, RaceFuzzer$_2$, and RaceFuzzer$_3$ do. And so on.

Of course, giving RaceFuzzer the benefit of access in advance to all test cases is overly generous, but this experiment serves to illustrate how the tool works. In contrast, RaceMob achieves data race detection coverage proportional to the number of runs with different inputs in_1, in_2, in_3, irrespective of which test cases were available initially, since it performs static race detection to identify potential races. RaceFuzzer could potentially miss all input-dependent races even when the program under test is run with the inputs that expose such races, because it may have missed those races in its initial instrumentation stage. However, this is not a fundamental shortcoming: it is possible to mitigate it by replacing RaceFuzzer's dynamic data race detection phase with a static data race detector.

The results of the comparison with Portend appear in Table 5. Portend discovered all the input-dependent races in bench$_4$, but failed to detect the races in the other benchmarks, because it employs a precise dynamic detector. RaceMob detects all three test cases for bench$_4$, as well as all the races in all the other benchmarks.

5.5 Scalability with Application Threads

RaceMob uses atomic operations to update internal shared structures related to dynamic data race validation and signal-wait synchronization to perform schedule steering; in this section, we analyze the effect these operations have on RaceMob's scalability as the number of application threads increases.

We configured multiple clients to concurrently request a 10 MB file from Apache and Knot using the Apache benchmarking tool `ab`. For SQLite and Memcached, we inserted, modified, and removed $5,000$ items from the database and the object cache, respectively. We used Pbzip2 to decompress a 100 MB file. For Ocean, we simulated currents in a 256×256 ocean grid. For Barnes, we simulated interactions of $16,384$ bodies (default number for Barnes). We varied the number of threads from 2 – 32. For all programs, we ran the instrumented versions of the programs while performing data race detection and measured the overhead relative to uninstrumented versions on the 8-core machine.

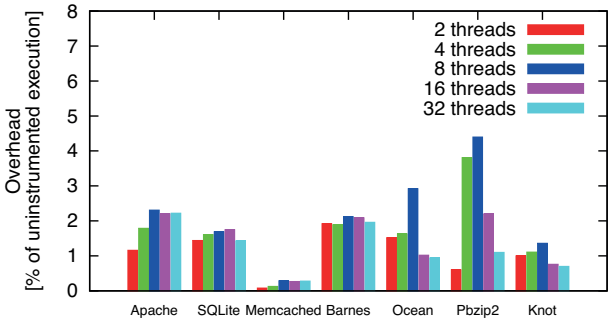

Figure 9: RaceMob scalability: Induced overhead as a function of the number of application threads.

Fig. 9 shows the results. We expected RaceMob's overhead to become less visible after the thread count

reached the core count. We wanted to verify this, and that is why we used the 8-core machine. For instance, for Apache the overhead is 1.16% for 2 threads, it slightly rises to its largest value of 2.31% for 8 threads, and then it decreases as the number of threads exceeds the number of cores. We observe a similar trend for all other applications. We conclude that RaceMob's runtime overhead remains low as the number of threads in the test programs increases.

6 Discussion

In this section we discuss some remaining open questions and RaceMob's limitations.

Why would application users want to use RaceMob? RaceMob can make the life of some users harder: it introduces some overhead (in our opinion negligible), and the use of schedule steering (§3.2.3) may trigger races with unpleasant effects that would otherwise not occur (although schedule steering is a configurable option). On the upside, once a user encounters a true race, the developer immediately learns about it and can fix it, instead of having to wait for users to report it and then spend time diagnosing it (which can take on the order of weeks [16]). This benefits the entire user community, and it amortizes the cost of detection across all members. For individual motivation, we envision developers offering rewards to users who find true races, similar to how users are rewarded in distributed DES and RSA cracking efforts or in Folding@home [1] or SETI@home [2]. Rewards could consist of free upgrades or trials, or "badges of honor" in the case of open-source software. RaceMob can additionally be employed for beta testers (e.g., Windows 7 had over 8 million beta testers [3]).

Per-user vs. aggregate overhead In order for the user community to get the benefits outlined above, it expends *in aggregate* a fair amount of CPU time and energy, even though to the individual user this is likely negligible. It is unclear how to quantify this trade-off, when this expenditure is set against the costs of delayed fixing of data race bugs. For example, multiple users losing data due to a given race may expend more energy recovering their lost work than participating in RaceMob. Some failures, such as security breaches using a race-based exploit [47], may have exorbitant cost. We do expect aggregate overhead for a given program to reduce over time and asymptotically reach zero, as data races are progressively eradicated from the software, so perhaps the trade-off is favorable to RaceMob in the long run.

Will RaceMob replace in-house testing? We view RaceMob as complementing existing testing techniques and making them more potent. For example, RaceMob can augment both the code and schedule coverage obtained by in-house testing, by "harvesting" executions that emerge naturally during real use but that are not part of the developers' test suite.

Will RaceMob encourage developers to release buggy software? We doubt developers would start relying on RaceMob to find bugs they could otherwise find on their own. Today developers release software with data race bugs not because they have an incentive to do so, but because they lack the right tools to productively find these bugs. We believe most developers want to improve their code, and RaceMob offers them help in deploying software that is free of data race bugs.

There is an exponentially large space of program states and schedules; can users' hardware make a difference? RaceMob's static race detection phase substantially reduces the exponential search space in code paths and possible schedules. In fact, the space that users' machines need to search is related more to the number of race candidates output by the static detector than by the size of the program. Note that RaceMob does not have to explore all possible paths and thread schedules, but rather only those that are relevant: the ones experienced by real users. Finally, an application's user population typically has a lot more hardware at its disposal than what the developer can devote to testing, especially with the rise of mobile devices (by the end of this year, the number of mobile devices is expected to surpass the size of the human population [8]).

Which are the limitations of RaceMob? Like any race detector, RaceMob must be explicitly made aware of all synchronization constructs, otherwise it may report legitimate synchronization constructs (e.g., the use of lock-free algorithms) as races. RaceMob recognizes the ad-hoc synchronization mechanisms described in [46].

A crowdsourced framework like RaceMob has certain privacy implications. For the case of data race detection, we believe these implications are minimal, but we are nevertheless looking into ways of quantifying the balance between privacy and the amount of execution information sent from users to the hive. Enabling an application to be controlled remotely by the hive may be problematic as well, though the developer (who operates the hive) is typically trusted by the users of his/her code.

Finally, RaceMob has no provisions for thwarting malicious users who send false results to the hive in order to fool it. One possibility is to cross-check results across

multiple reporting users, to achieve some statistical confidence in their veracity. Alternatively, the hive could "test" individual users by sending validation tasks for races it already has a certain verdict for, and checking if the user validation result is correct; dishonest users can be removed from the system. Other options include trusting only verified user accounts (as in the case of verified beta testers) as well as using various reputation systems.

7 Related Work

RaceMob's crowdsourcing approach is in part inspired by cooperative bug isolation (CBI) [28]. CBI collects various information about program execution at user sites from both failing and successful runs to identify the likely causes of failures. The first system that detected concurrency bugs in a collaborative setup was CCI [21]. CCI extended CBI to gather information pertaining to concurrency bugs. Both CBI and CCI detect bugs that cause software failures. RaceMob targets a different problem: detecting all data races in a program, most of which only rarely cause visible failures.

RaceMob is inspired in part by Windows Error Reporting (WER) [15], a large collaborative error reporting system developed by Microsoft. WER collects information (e.g., coredumps) after a crash, in order to prioritize potential bugs. In some sense, WER implicitly crowdsources program executions and gathers reports after crashes, then formulates a hypothesis about a potential bug. This hypothesis has to be validated manually. RaceMob reverses this process: it formulates the hypothesis prior to crowdsourcing (based on static analysis) and then uses crowdsourcing to automatically validate it.

Crowdsourcing has the same effect as sampling, in that it reduces the runtime overhead either by performing data race detection using temporal sampling (for a certain time interval, e.g., as in PACER [6]), or using spatial sampling (for certain accesses in the program, e.g., as in LiteRace [31]). RaceMob achieves spatial sampling by crowdsourcing and temporal sampling by on-demand data race detection. RaceMob's static race detection phase further improves upon traditional temporal sampling by allowing RaceMob to determine when to start sampling.

Prior research combined static and dynamic analysis to perform data race detection. Goldilocks [9] and Choi et al. [7] used a static thread escape analysis phase to eliminate the need to track thread-local variables. RaceMob takes a similar approach to these tools, but uses a complete detector—which is more accurate than just using thread escape analysis—to detect *all* data races. It then uses on-demand data race detection and crowdsourcing to achieve lower runtime overhead than these tools.

Schedule steering was previously used to analyze the consequences of known data races (i.e., data races previously detected using dynamic detectors) [33, 39, 24]. However, schedule steering by itself is not enough to find a data race—either because the accesses may need to happen closer together in time, or because the accesses will only race in particular environments and inputs. RaceMob focuses on accurately detecting data races (rather than just analyzing their consequences) by using on-demand data race detection in addition to schedule steering, and uses static race detection results as hints for which memory accesses to reorder during race detection.

Exterminator [35] was the first system to propose collaborative bug fixing for memory errors. Aviso [30] proposed a collaborative approach for fixing concurrency bugs. Although RaceMob focuses on collaborative detection instead of fixing, it could be integrated with CFix [22], an automated concurrency bug fixing tool, to avoid racing thread interleavings.

8 Conclusion

In this paper, we described RaceMob, a crowdsourcing-based approach for better data race detection: it employs static analysis to find candidates for data races, and then dynamically validates these candidates by leveraging execution information from all its users. This two-phase static–dynamic approach for detecting data races was shown to work on several real-world programs and systems; it has higher accuracy and lower overhead than the state of the art. We described a new algorithm for dynamically validating data races on-demand, and a crowdsourcing framework that, unlike traditional testing, taps directly into real user executions to detect data races. Overall, RaceMob detected 106 real data races in ten programs, with an average runtime overhead of 2.32%.

Acknowledgments

We are indebted to our shepherd Rebecca Isaacs, to Madan Musuvathi and Jon Howell, to the anonymous reviewers, and to Silviu Andrica, Edouard Bugnion, and Volodymyr Kuznetsov for their insightful feedback and generous help in improving this paper. This work was supported in part by ERC Starting Grant No. 278656 and by gifts from Google and Intel.

References

[1] Folding@home. http://folding.stanford.edu.
[2] SETI@home. http://setiathome.berkeley.edu.

[3] A history of Windows. http://windows.microsoft.com/en-us/windows/history, 2013.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Intl. Conf. on Compiler Construction*, 2004.

[5] H.-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX Workshop on Hot Topics in Parallelism*, 2011.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Intl. Conf. on Programming Language Design and Implem.*, 2010.

[7] 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. *SIGPLAN Notices*, 37(5):258–269, 2002.

[8] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2012-2017. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.

[9] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Intl. Conf. on Programming Language Design and Implem.*, 2007.

[10] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symp. on Operating Systems Principles*, 2003.

[11] P. Eriksson. Parallel file scanner. http://ostatic.com/pfscan, 2013.

[12] B. Fitzpatrick. Memcached. http://memcached.org, 2013.

[13] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Intl. Conf. on Programming Language Design and Implem.*, 2009.

[14] J. Gilchrist. Parallel BZIP2. http://compression.ca/pbzip2, 2013.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.

[16] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *Intl. Conf. on Computer Aided Verification*, 2008.

[17] Apache httpd. http://httpd.apache.org, 2013.

[18] Intel Corp. Parallel Inspector. http://software.intel.com/en-us/articles/intel-parallel-inspector, 2012.

[19] ISO/IEC 14882:2011: Information technology – programming languages – C++. International Organization for Standardization, 2011.

[20] ISO/IEC 9899:2011: Information technology – programming languages – C. International Organization for Standardization, 2011.

[21] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. *SIGPLAN Not.*, 2010.

[22] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Symp. on Operating Sys. Design and Implem.*, 2012.

[23] S. B. John Erickson, Madanlal Musuvathi and K. Olynyk. Effective data-race detection for the kernel. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[24] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with Portend. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[27] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Intl. Conf. on Programming Language Design and Implem.*, 2012.

[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[29] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – A comprehensive study on real world concurrency bug characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[30] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[31] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Intl. Conf. on Programming Language Design and Implem.*, 2009.

[32] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Intl. Conf. on Programming Language Design and Implem.*, 2006.

[33] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. *Intl. Conf. on Programming Language Design and Implem.*, 2007.

[34] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer.

CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Construction*, 2002.

[35] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Intl. Conf. on Programming Language Design and Implem.*, 2007.

[36] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symp. on Principles and Practice of Parallel Computing*, 2003.

[37] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Symp. on Principles and Practice of Parallel Computing*, 2003.

[38] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[39] K. Sen. Race directed random testing of concurrent programs. *Intl. Conf. on Programming Language Design and Implem.*, 2008.

[40] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer - Data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[41] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. Technical Report CSL-TR-92-526, Stanford University Computer Systems Laboratory, 1992.

[42] SQLite. http://www.sqlite.org/, 2013.

[43] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Symp. on Operating Systems Principles*, 2003.

[44] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *Symp. on the Foundations of Software Eng.*, 2007.

[45] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Intl. Conf. on Programming Language Design and Implem.*, 1995.

[46] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad-hoc synchronization considered harmful. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[47] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *USENIX Workshop on Hot Topics in Parallelism*, 2012.

[48] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Intl. Symp. on Computer Architecture*, 2009.