Effective Sampling for Lightweight Data-Race Detection

Daniel Marino
University of California, Los Angeles
dlmarino@cs.ucla.edu

Madanlal Musuvathi
Microsoft Research, Redmond
madanm@microsoft.com

Satish Narayanasamy
University of Michigan, Ann Arbor
nsatish@umich.edu

Abstract

Data races are one of the most common and subtle causes of pernicious concurrency bugs. Static techniques for preventing data races are overly conservative and do not scale well to large programs. Past research has produced several dynamic data race detectors that can be applied to large programs and are precise in the sense that they only report actual data races. However, these dynamic data race detectors incur a high performance overhead, slowing down a program's execution by an order of magnitude.

In this paper we present FeatherLite, a very lightweight data race detector that samples and analyzes only selected portions of a program's execution. We show that it is possible to sample a multi-threaded program at a low frequency and yet find infrequently occurring data races. We implemented FeatherLite using Microsoft's Phoenix compiler. Our experiments with several Microsoft programs show that FeatherLite is able to find more than 75% of data races by sampling less than 5% of memory accesses in a given program execution.

1. Introduction

Multi-threaded programs are notoriously difficult to understand and debug, largely due to the non-deterministic way in which instructions from the various threads of execution may be interleaved. As a result, even well-tested concurrent programs contain subtle bugs that may not be discovered until long after deployment. Data races [28] are one of the most common sources of bugs in shared-memory, multi-threaded programs. A data race happens when multiple threads perform conflicting data accesses without proper synchronization. The effects of a data race range from subtle memory corruption issues to exposure to unexpected memory model effects of the underlying compiler [23, 4] and hardware [1].

Over the last couple of decades, several static and dynamic techniques have been developed to automatically find data races in a multi-threaded program. Static techniques [5, 15, 33, 37, 17, 34, 14, 40, 27] are complete, but are not scalable. Also, they tend to make conservative assumptions that lead to a large number of false data races. On the other hand, dynamic techniques [38, 30, 42, 13] are scalable and more precise than static tools. Though they are not complete, coverage can be increased through more tests. A severe limitation of dynamic data races detectors, however, is their runtime overhead. Data race detectors like RaceTrack [42] that are implemented as part of managed runtime system, incur about 2x to 3x slowdown. Data race detectors for unmanaged programs like Intel's Thread Checker [36], incur performance overhead in the order of 200x. Such a high overhead prevents the use of dynamic data-race detectors in practice. For instance, programmers do not enable heavy-weight analysis like data-race detection during beta-testing of industrial applications. The reason is that these tools not only hog the resources, but also do not allow the programmers to test realistic program executions.

The main reason for such a high performance overhead for these tools is that data-race detection requires analyzing every memory operation executed by the program. In this paper, we propose to use *sampling*. By processing only a small percentage of memory accesses, a sampling-based approach can significantly reduce the runtime overhead of data-race detection.

At the outset, however, a sampling-based data-race detector seems like a non-starter. Most memory accesses do not participate in data races. Sampling approaches, in general, have difficulty capturing such rare events. To make matters worse, a data race results from two conflicting accesses. Thus, the sampler has to capture *both* of the accesses in order to detect the data race. Due to the multiplicative effect of sampling probabilities, a naive sampling algorithm will fail detect most of the data races.

We present a sampling algorithm for effective data-race detection. The sampling algorithm is based on the *cold-region hypothesis* that data races occur when a thread is executing a "cold" (infrequently accessed) region in the program. Data races that occur in hot regions of well-tested programs are either already been found and fixed, or such races are likely to be benign. Our adaptive sampler starts off by sampling all the code regions at 100% sampling rate. But every time a code region is sampled, its sampling rate is progressively reduced until it reaches a lower bound. Thus, cold regions are sampled at a very high sampling rate, while adaptively reducing the rate for hot regions to a very small value. Since a program spends most of its time executing hot regions, the adaptive sampler automatically avoids slowing down performance-critical hot regions of the program.

This paper describes an implementation of the sampling algorithm in a tool called FeatherLite, and demonstrates its effectiveness on a wide range of programs. FeatherLite is implemented using the Phoenix compiler [24] for statically rewriting (x86) program binaries. For every function, FeatherLite produces an instrumented copy of the function that logs all memory accesses and synchronization operations in the function. At runtime, FeatherLite switches the execution between the uninstrumented and instrumented functions based on the sampling information maintained per thread. Our implementation is an extension of the adaptive profiling technique in SWAT [16]. The key difference is that our sampler needs to be "thread-aware". Otherwise, a code region might become hot when a thread executes it many times, and then when the same code region is executed for the first time in a new concurrent thread, it might not get sampled as it would be incorrectly treated as a hot region. Thus, FeatherLite maintains profiling information per thread. To our knowledge, FeatherLite is the first data-race detection tool that uses sampling to reduce the runtime performance cost.

This paper describes many of the challenges and tradeoffs involved in building a tool like FeatherLite. One of our key requirements of FeatherLite is that it never reports a false data race. Data races, like many concurrency bugs, are very hard to debug. We deemed it unacceptable for the users of the tool to spend lots of time triaging false error reports. Thus, FeatherLite, while sampling memory accesses, still captures *all* the synchronizations in the program, which is necessary to ensure that there no false positives.

Even if only a portion of a program's execution is analyzed, an online dynamic data race detector would still have to keep track of meta-data for each memory location. Tracking meta-data could easily become a source of performance bottleneck. To avoid this cost, and reduce the cost of the data race check itself, the proposed solution executes instrumented code only for the sampled memory operations and records them in a log. In addition, all the synchronization operations are also logged. The log could be consumed either by an online data race detector executing concurrently on a spare processor-core in a many-core processor, or by an offline data race detector. In this paper, we focus on the latter. The offline data race detector could be either a happens-before based [19] or a lockset based detector [38]. But, we chose the former as it does not report false races.

We present following contributions in this paper:

- We demonstrate that the technique of sampling can be used to significantly reduce the runtime overhead of a data race detector without introducing any false positives. FeatherLite is the first data-race detection tool that uses sampling to reduce the runtime performance cost. As FeatherLite permits users to specify a bound on the performance overhead, we expect that such a sampling-based approach will encourage users to enable data race detection even during beta-testing of industrial applications.
- We discuss several sampling strategies, and show that choosing an appropriate strategy is essential to maintaining a high detection rate while keeping sampling rate low. In particular, we show that an adaptive sampler that heavily samples the first few executions of a function *in each thread* is effective.
- We implemented FeatherLite using the Phoenix compiler, and used it to analyze several Microsoft programs such as ConcRT, Dryad and several micro-benchmarks. The results show that by logging less than 5% of memory operations, we can detect more than 75% of data races in a particular execution.

The rest of this paper is organized as follows. In Section 2 we review happens-before data race detector and the reasons for its high runtime overhead. Section 3 presents an overview of our sampling based approach to reduce the runtime cost of a data race detector. Section 4 details the implementation of our race detector. We present our experimental results in Section 5. In Section 6 we describe related work and position our contributions, and conclude in Section 7.

2. Background

Dynamic data race detectors [38, 42] are precise and scalable to large applications. However, they incur high runtime overhead unmanaged programs (Intel's ThreadChecker [36], for instance, incurs nearly 200x slowdown), which we seek to address in this paper. Dynamic data race detectors can be classified into two major categories: happens-before based and lockset based. Happens-before data race detectors [19, 9] find only the data races that manifest in a given program execution. Lockset based techniques [38] can predict data races that have not manifested in a given program execution, but can report false positives. In this work, we focus on happens-before based data race detectors as they do not report any false positives. However, our approach to sampling could equally well be applied to a lockset algorithm.

In this section we review how happens-before race detection works and the reasons for the runtime overhead of a happens-before data race detector.

2.1 Happens-Before Race Detection

Here we provide a brief review of detecting data races by using the happens-before relation on program events. The happens-before relation, $\longrightarrow$, is a partial order on the events of a particular execution of a (multi-threaded) program. It can be defined by the following rules:

(HB1) $\mathtt{a} \longrightarrow \mathtt{b}$ if `a` and `b` are events from the same sequential thread of execution and `a` executed before `b`.

(HB2) $\mathtt{a} \longrightarrow \mathtt{b}$ if `a` and `b` are synchronization operations from different threads such that the semantics of the synchronization dictates that `a` precedes `b`.

(HB3) The relation is transitive, so if $\mathtt{a} \longrightarrow \mathtt{b}$ and $\mathtt{b} \longrightarrow \mathtt{c}$, then $\mathtt{a} \longrightarrow \mathtt{c}$.

We can then define a data race as a pair of accesses to the same memory location, where at least one of the accesses is a write, and neither one happens-before the other. One advantage that happens-before race detection has over the lockset-based approach is that, it supports a wide range of synchronization paradigms, and not just mutual exclusion locks. For instance, our formulation of the second rule for defining happens-before allows us to introduce a happens-before ordering between a call to `fork` in a parent thread and the first event in the forked child thread.

Figures 1 and 2 show how the happens-before relationship is used to find data races. The edges between instructions indicate a happens-before relationship derived using

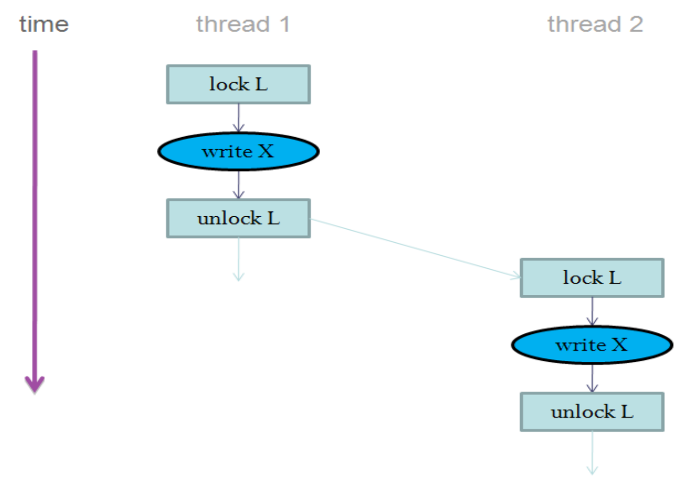

Figure 1. Example of properly synchronized accesses to a memory location X. Edges between nodes represent a happens-before relationship. There is no data race in this case because there is a path from write into X to the other.

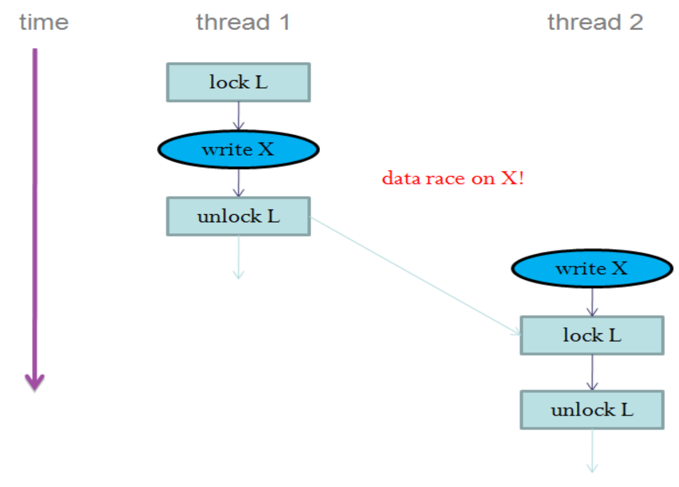

Figure 2. Example of an *im*properly synchronized access to a shared memory location X. There is no happens-before path from either write to the other, thus we have a data race.

rule HB1 or HB2. Transitively, by HB3, any two nodes with a path between them are also in a happens-before relationship. Figure 1 shows an example of two properly synchronized accesses to a shared memory location. Since the two writes have a path between them, they do not race with each other. In Figure 2, thread 2 accesses the shared memory location without proper synchronization. Because there is no path between the two writes, we have a data race in this case.

2.2 Source of Runtime Overhead

There are two primary sources of overhead for a happens-before dynamic data race detector. One, it needs to instrument all the memory operations and all the synchronizations operations executed by the application. This results in a high performance cost due to the increase in the number of instructions executed at runtime. Two, it needs to maintain meta-data for each memory location accessed by the application. Most of the happens-before based algorithms [19, 28, 2, 7, 8, 10, 9, 39, 31, 35, 26] use vector clocks to keep track of the times of all the memory operations along with the address locations they accessed. Maintaining such meta-data increases memory overhead.

3. FeatherLite Overview

This section presents a high-level overview of FeatherLite. The implementation details together with various design trade-offs are discussed in Section 4.

FeatherLite has two key goals. First, FeatherLite should not add too much runtime overhead during dynamic data-race detection. Our eventual goal is to run FeatherLite during beta-testing of industrial applications. Prohibitive slowdown of existing detectors limits the amount of testing that can be done for a given amount of resources. Also, users shy away from intrusive tools that do not allow them to test realistic program executions. Second, FeatherLite should never report a *false* data race. Data races are very difficult to debug and false positives severely limit the usability of a tool from the users' perspective. This goal, surprisingly, has decided many of our design decisions in FeatherLite.

3.1 Case for sampling

The key premise behind FeatherLite is that sampling techniques can be effective for data-race detection. While reducing the runtime overhead, the main trade-off of a sampling approach is that it can miss data races. We argue that this trade-off is acceptable for the following reasons. First, dynamic techniques cannot find *all* the data races in the program, anyway. They can only find data races based on the interleavings explored by the program in one particular execution. Furthermore, a low-overhead sampling based data race detector would encourage users to use it on lot more executions of the program, possibly achieving better coverage. For instance, programmers can enable data race detectors during beta-tests at the customer site, or even the production software can be shipped to users with sampling turned on [21], thereby enabling data-race detection in the wild, an advantage existing detectors cannot possibly achieve.

Another key advantage of sampling techniques is that they provide a useful *knob* that users can use to trade runtime overhead for coverage. Such control is currently not possible for existing data race detectors. For instance, while testing interactive applications that spend most of their time waiting for the user input, a programmer can increase the sampling rate, because the latency of race detection is likely to be masked by the I/O latency of the tested application.

3.2 Logging Events

Data-race detection requires logging the following events at runtime.

- reads and writes to memory, logged in the program order,

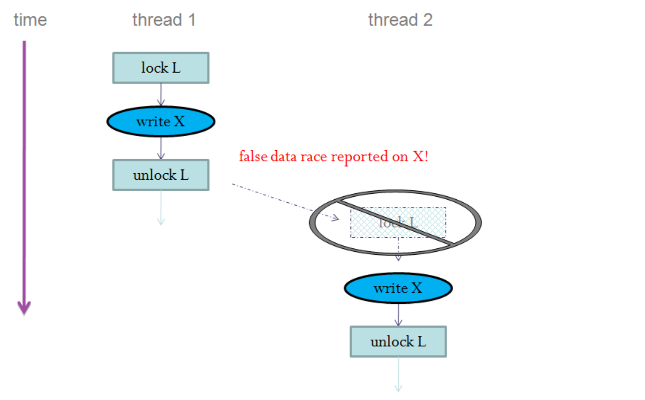

Figure 3. Failing to log a synchronization operation results in loss of happens-before edges. There is no data race in this execution, but a false data race on X would be reported.

- synchronization operations along with a time-stamp that allows us to construct the happens-before ordering between synchronization operations executed in different threads.

These logs can then be analyzed offline or during program execution (§4.4). A data race is detected, if there is no synchronization ordering between two pair of accesses to the same memory location, and at least one of them being a write.

3.3 Using Sampling

Clearly adding code to log every memory access imposes a significant overhead. By sampling only a fraction of these events we can reduce the overhead in two ways. First, the execution of the program is much faster because of the reduced instrumentation. Second, the offline data-race detection algorithm needs to process less number of events making it faster as well.

While we seek to reduce the runtime overhead using sampling, we must be careful in choosing which events to log and which events not to log. In particular, we have to log *all* synchronization events in order to avoid reporting false data races. Figure 3 shows why this is the case. Synchronization operations induce happens-before orderings between program events. Any missed operation can result in missing edges in the happens-before graph. The data-race detection algorithm will therefore incorrectly report races on accesses that are otherwise ordered by the unlogged synchronization operations. To avoid such false positives, it is necessary to log all synchronization operations. However, for most applications, proportion of the number of synchronization operations when compared to the number of instructions executed in a program is small, and therefore does not cause significant performance overhead.

We can, however, selectively sample the memory accesses. If we choose not to log a particular memory access, we risk missing a data race involving that access (a false negative). As we discussed in Section 3.1, this is an acceptable trade-off. But for this to be a reasonable trade-off, choosing a good strategy for selecting memory accesses to log is essential. Data race involves two accesses and a sampler needs to successfully log both of them for detecting a race. We describe such a sampler below.

3.4 Sampler Implementation

In this paper, we treat every function as a code region. Our static instrumentation tool creates two copies for each function as shown in Figure 4. The instrumented function logs all the memory operations (their addresses and program counter values) and synchronization operations (memory addresses of the synchronization variables along with their timestamps) executed in the function. The un-instrumented copy of the function logs only the synchronization operations. Before entering a function, the sampler (represented as dispatch check in Figure 4) is executed. Based on the decision of the sampler, either the instrumented copy or the un-instrumented copy of the function is executed. As the dispatch check happens once per every function call, we have to make the dispatch code as efficient as possible.

3.5 Thread Local Adaptive Bursty Sampler

There are two requirements for a sampling strategy. Ideally, a sampling strategy should maintain a high data-race detection rate even with a low sampling rate. Also, it should enable an efficient implementation of the dispatch check that determines if a function should be sampled or not.

Our sampler is an extension of the adaptive bursty sampler [16], previously shown to be successful for detecting memory leaks. An adaptive bursty sampler starts off by analyzing a code region at 100% sampling rate, which means that the sampler would always invoke the instrumented copy of the function. But every time when a code region is executed, its sampling rate is reduced by a factor $Decr$, until the sampling rate reaches a lower bound L. The sampler is also bursty, which means, for each sample, a code region is profiled for $numProfiles$ consecutive instances of its execution. This ensures that, frequently executed "hot" code regions are sampled at much lower frequency than infrequently executed "cold" regions in a program. This sampling algorithm is based on the hypothesis that data races, at least in reasonably well-tested programs, occur when a thread executes a cold region. Data-races between two hot paths are unlikely — either such a data race is already found during testing and fixed, or such a data race is likely to be a *benign* or intentional data race.

To make adaptive bursty sampler effective for data-race detection, we extend the above algorithm by making it "thread aware". One problem with the "global" adaptive bursty sampler [16] is that a particular function can be considered "hot" even when a thread executes the function for the first time. This is because other threads might have executed the same function many times. The sampler in FeatherLite is a "thread local" adaptive bursty sampler. It maintains separate sampling information in each thread. Our experiments (§5) show that this extension significantly improves the effectiveness of FeatherLite.

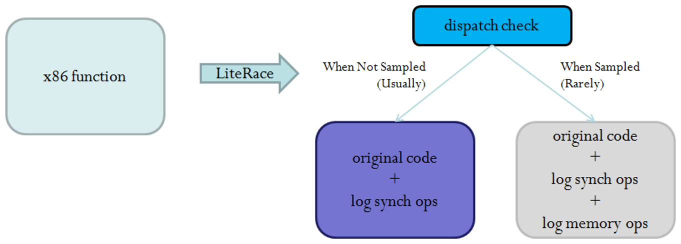
x86 function
LiteRace
dispatch check
When Not Sampled (Usually)
When Sampled (Rarely)
original code + log synch ops
original code + log synch ops + log memory ops

Figure 4. FeatherLite Instrumentation.

Table 1. Logging synchronization operations.

Synchronization Op	SyncVar	Add'l Sync Required?
Lock / Unlock	Mutex	No
Signal / Wait	Event	No
Fork / Join	Child Thread Id	Yes
Atomic Machine Ops	Memory Location	Yes

4. FeatherLite Implementation

This section describes the implementation details of FeatherLite. It is made up of two components: a profiler built using a binary instrumentation tool that inserts code into the target application to log events of interest during a program's execution, and a happens-before based data race detector that processes the logged events from an execution of the instrumented application as input, and produces a report of the data races found.

4.1 Instrumenting the Code

FeatherLite is based on static instrumentation of x86 binaries. To ensure the practical usability of our tool, it is important that FeatherLite works on binaries without requiring the source code. We use the Phoenix [24] compiler and analysis framework to parse the x86 executables to perform the transformation depicted in Figure 4. FeatherLite instruments a program to create two versions for each function: one that logs all the memory operations and another that does not log any memory operation. As explained in Section 3, avoiding false positives requires logging synchronization operations in both versions of the function. A dispatch check at function entry decides which version of the function should be used during a particular dynamic call of a function.

As the dispatch check in Figure 4 is executed every time the function is executed, it is important to keep this overhead low. In contrast to prior adaptive sampling techniques [16], FeatherLite maintains profiling information *per thread*. For each thread, FeatherLite maintains a buffer in the thread local storage. These buffers contain two counters for each instrumented function. One keeps track of the number of times a function that the thread executed, and another called sampling counter that is used to determine when to sample next. On function entry, the dispatch check decrements the sampling counter corresponding to that function maintained in the thread local buffer. If the sampling counter's value is non-zero, the light-weight copy of the function is invoked. When the sampling counter reaches zero, the dispatch check invokes the fully instrumented version of the function (one that logs memory operations), and sets the sampling counter to a new value based on the current sampling rate for the function. The current sampling rate for the function is determined based on the number of times it got executed by the thread (kept track of by the frequency counter).

For efficient access to these buffers, FeatherLite avoids function calls into standard APIs for accessing thread-local storage. Instead, we implemented our own version for the thread-local resolution using the *Thread Execution Block* [25] structure maintained by the Windows OS for each thread. Also, the dispatch check uses a single register `edx` for its computation. The instrumentation tool analyzes the original binary for the function to check if this register and the `eflags` register are live at function entry, and injects code to save these registers only when necessary. In the common case, our dispatch check involves 8 instructions with 3 memory references and 1 branch (that is mostly not taken). We measure the runtime overhead of the dispatch check in Section 5.

4.2 Tracking Happens-before

As mentioned earlier, avoiding false positives requires accurate happens-before data. Ensuring that we correctly record the happens-before relation for events of the same thread is trivial since the logging code executes on the same thread as the events being recorded. Correctly capturing the happens-before data induced by the synchronization operations between threads in a particular program execution requires more work.

We associate with each synchronization operation a *syncVar*. As an example, for a `lock` operation, the syncVar is the address of the mutex being locked. Table 1 shows the syncVar used for some of the synchronization operations that FeatherLite supports. For each synchronization operation executed in the target program, we log the associated syncVar along with a logical timestamp which we increment. We can then say that if `a` and `b` are two operations on the same syncVar and `a` has a smaller logical timestamp than `b`, then $\texttt{a} \longrightarrow \texttt{b}$.

In order for the timestamps to be valid, we must ensure that for all operations on a particular syncVar, the logical timestamp reflects the actual (real time) order in which the operations occurred. For some kinds of synchronization, we are able to leverage the semantics of the operation to guarantee that the logged timestamps are consistent with the actual order. For instance, by logging and incrementing the timestamp *after* a `lock` instruction and *before* an `unlock` instruction, we guarantee that an `unlock` operation on a particular mutex will have a smaller timestamp than a subsequent `lock` operation on that same mutex in another thread. Note that if we incremented the timestamp after an `unlock` operation, it would be possible for another thread to execute a `lock` on the same mutex and receive a lower logical timestamp than the one eventually given to the preceding `unlock`. It would then appear as if accesses within these two critical section could have been executed concurrently causing false data races to be reported.

Whenever possible, we avoid introducing synchronization not present in the original program in order to achieve our goal of extremely low overhead. But sometimes we must add our own synchronization to correctly record the happens-before ordering and avoid false positives. For example, consider a target program that uses atomic compare and exchange instructions to implement its own locking. Since we don't know if a particular compare and exchange is acting as a "lock" or as an "unlock", we must introduce a critical section to avoid interleavings that would allow two exchanges to the same memory location to be logged with timestamps inconsistent with the actual ordering of the events.

4.3 Handling Dynamic Allocation

Another subtle issue is that a dynamic data-race detector should account for the reallocation of the same memory to a different thread. A naive detector might report a data-race between accesses to the reallocated memory with accesses perform during a prior allocation. To avoid such false positives, FeatherLite additionally monitors all memory allocation routines and treats them as additional synchronization performed on the memory page containing the allocated or deleted memory.

4.4 Analyzing the Logs

The FeatherLite profiler generates a stream of logged events during program execution. In our current implementation, we write these events to the disk and process them offline for data races. Our main motivation for this design decision was to minimize perturbation the runtime execution of the program. We are also currently investigating an online detector that can avoid runtime slowdown by using an idle core in a many-core processor.

The log events are processed using a standard implementation [36] of the happens-before based data-race detector, described in Section 2.1. We avoided the use of lock-set based data-race detection algorithms to ensure no false positives.

5. Results

In this section we present our experimental results. In Section 5.1 we show that our thread-local adaptive sampler achieves a high race detection rate while maintaining a low sampling rate. We compare it with several other samplers. Section 5.2 discusses the performance of our implementation on a several benchmarks. All experiments were run on an Intel Core 2 Duo 3.0 GHz processor and 4 GB of RAM running Windows Vista.

5.1 Sampling Strategy Comparison

In this section, we compare different samplers and show that the thread-local adaptive sampler outperforms other samplers. For our benchmarks, our sampler is able to find, on average, more than 75% of data races while sampling less than 5% of memory accesses in a given execution.

To have a fair comparison, different samplers need to be evaluated on the *same* thread interleaving of the program. However, two different executions of a multi-threaded program is not guaranteed to yield the same interleaving even if the input is the same. To compare the effectiveness of various samplers in detecting data races accurately, we created a modified version of FeatherLite that performs full profiling, where all synchronization and all memory operations are logged. In addition to the complete logging, on each function entry, we execute the "dispatch check" logic from each of the samplers we wish to compare. We then mark in the log whether or not each of the samplers would have logged a particular memory operation.

By performing race detection on the complete log, we find all the data races that happened during the program's execution. We can then perform race detection on the subset of the log corresponding to the operations a particular sampler would have recorded. Then, by comparing the results with those from the complete log, we are able to calculate the proportion of data races detected (called detected rate) by each of the samplers.

The samplers that we evaluate are listed in Table 2. The "Short Name" column shows the abbreviation we will use for the sampler in the figures. Two averages for effective sampling rate are shown. These are measures of the actual percentage of memory operations that are logged by each of the samplers, averaged over the five benchmarks. The weighted average weighs the benchmarks based on the number of memory operations performed.

FeatherLite's thread-local adaptive sampler is the first one listed in the table. For each thread and for each function, the sampler starts with a 100% sampling rate progressively reducing the rate till it reaches a base sampling rate of 0.1%. To determine the effect of this adaptive backoff, our second sampler uses a fixed 5% sampling rate per thread per function. The next two samplers are "global" versions of the two samplers above and backoff based on the executions of a function irrespective of the calling thread. The global adaptive sampler is similar to the one used in SWAT [16], except with an increased sampling rate. Even with this increased rate, our experiments show that the global samplers are not as effective as the thread-local samplers at finding data races. The random samplers log each function call at random. The final sampler evaluates the cold-region hypothesis by logging only the "uncold" regions — it logs all but the first 10 calls of a function per thread.

For our benchmarks, we selected two industrial-scale concurrent programs. Dryad is a distributed execution engine for coarse-grained data-parallel applications [18]. The test harness for Dryad for our experiments was provided by its lead developer. The test exercises the shared-memory channel library used for communication between the nodes in Dryad. We tried two versions of Dryad, one with the standard C library statically linked in and the other without. For the former, FeatherLite instruments all the standard library functions called by Dryad. ConcRT is a concurrent runtime library that provides lightweight tasks and synchronization primitives for data-parallel applications. It is part of the parallel extensions to the .NET framework [12]. We used three different tests, namely Messaging, Agents, and Explicit scheduling, all part of the ConcRT concurrency test suite.

Table 2. Samplers that are compared along with the short name used in figures, a description, and the effective sampling rate for the benchmarks. The weighted average is weighted by the number of memory accesses in the benchmark application.

Sampling Strategy	Short Name	Description	Weighted Average ESR	Average ESR
Thread-local Adaptive	TL-Ad	Adaptive backoff per function / per thread (100%,10%,1%,0.1%)	2.3%	13.7%
Thread-local Fixed 5%	TL-Fx	Fixed 5% per function / per thread	6.6%	17.3%
Global Adaptive	G-Ad	Adaptive backoff per function globally (100%, 50%, 25%, ... , 0.1%)	0.9%	3.8%
Global Fixed	G-Fx	Fixed 10% per function globally	10.0%	10.6%
Random 10%	Rnd10	Random 10% of dynamic calls chosen for sampling.	9.9%	9.3%
Random 25%	Rnd25	Random 25% of dynamic calls chosen for sampling	24.6%	23.3%
Un-Cold Path	UCP	First 10 calls per function / per thread are NOT sampled, all remaining calls are sampled	98.0%	86.7%

Benchmarks	# data races found
Dryad Channel + stdlib	8
Dryad Channel	7
ConcRT Messaging	21
ConcRT Agents	21
ConcRT Explicit Scheduling	50

Table 3. Total number of data races found by FeatherLite.

For each of the benchmarks, we ran the application three times using our version of FeatherLite modified for sampler evaluation. The detection rate we report for each benchmark is an average rate of the three runs. The results for overall detection rate are shown in Figure 5, grouped by sampler with a bar for each benchmark within each group. The weighted average effective sampling rate for each of the samplers is also shown. Ideally, we would like a very low effective sampling rate and a very high detection rate. Notice that the FeatherLite sampler (TL-Ad) achieves this. The fixed rate thread-local sampler also performs well, but its effective sampling rate is higher than for FeatherLite's adaptive sampler. Another notable result from the figure is the fact that the "Un-Cold Path" sampler detects far fewer races than our sampler despite having an effective sampling rate approaching 100%.

During this experiment, Table 3 shows the total number of data races found for each benchmark. Details of the effective sampling rate for each of the samplers on each benchmark are given in Figure 6. The sampling rate of FeatherLite's sampler remains below 7% except for one benchmark, ConcRT Agents. This benchmark has many short-lived threads that don't repeatedly call the same functions. Thus, the thread-local samplers, which always sample the first few executions of each function in each thread, have a high sampling rate. This also explains why this is the only benchmark where the sampling rate for the un-cold path sampler drops below 90%.

5.1.1 Rare Data Race Detection

If the same data race occurs frequently, then it is likely that many sampling strategies would find them. However,

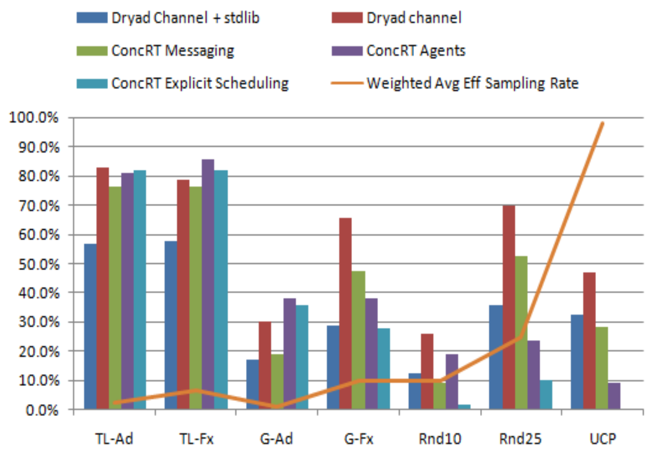

Figure 5. Data race detection rate for various samplers. The line shows average effective sampling rate for memory accesses.

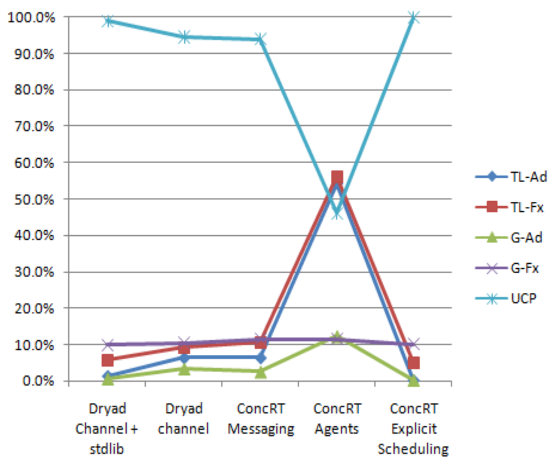

Figure 6. The effective memory access sampling rate for samplers across benchmarks.

it is more challenging to find data races that occur rarely at runtime. To quantify this, we classified all of the data races that were detected (using the full, unsampled log) based on the number of times that a race between the same 2 machine instructions was seen. We classify instruction pairs that raced three or fewer times as rare data races. The rest are considered frequent. Detection rates for these two categories are shown in Figure 7.

Most of the samplers perform well for the frequent data races. But, for infrequently occurring data races, the thread-local samplers are the clear winners. Note that the random sampler finds very few rare data races.

5.2 Analysis of Overhead

In Section 5.1 we presented results showing that the thread-local, adaptive sampler performs well. Here we present performance results from our implementation of FeatherLite, which uses this sampling strategy.

Apart from the benchmarks used in 5.1, we used additional compute and synchronization intensive benchmarks for our performance study. LKRHash is an efficient hash table implementation that uses a combination of lock-free techniques and high-level synchronizations. LKRHash is synchronization intensive. LFList is an implementation of a lock-free linked list available from [20]. LFList_Out is a modified version of LFList that performed additional computations apart from making linked list operations.

For each of the benchmarks, we ran FeatherLite 10 times in different configurations and took the average runtime. These configurations include, enabling just the dispatch check, then including the logging of synchronization operations, and finally including the sampled memory accesses. This allowed us to measure the overhead of the different components in FeatherLite.

Figure 8 shows the cost of using FeatherLite on the various benchmarks. Results for both real (clock) time and CPU time are given for each of the benchmarks. The real clock time is a better metric to evaluate the overhead of FeatherLite as it accounts for the overall system performance (by accounting for the time spent in accessing disk and other I/O operations). But we also show CPU time to illustrate the relative overhead of several components in FeatherLite.

The bottom portion of each vertical bar in Figure 8 represents the time it takes to run the uninstrumented application. The overhead incurred by the various components of FeatherLite are stacked on top of that. Except for LFList_Out and Dryad, other programs we analyzed are synchronization intensive. Therefore, the overhead is dominated by logging synchronization operations. As described in Section 3, avoiding false positives requires that we log all synchronization operations. The dispatch check results in a large proportion of the CPU overhead for Dryad. This suggests many calls to functions with few instructions.

On measuring the real time, we found a surprising but *repeatable* speed up of the Dryad application with FeatherLite. We believe this is due to the heavy interaction of the disk accesses in Dryad. In summary, FeatherLite adds negligible real time overhead in Dryad and LFList_Out. For synchronization intensive micro-benchmarks, the real time overhead is on the order of 2x to 2.5x, primarily dominated by the cost of logging synchronization operations.

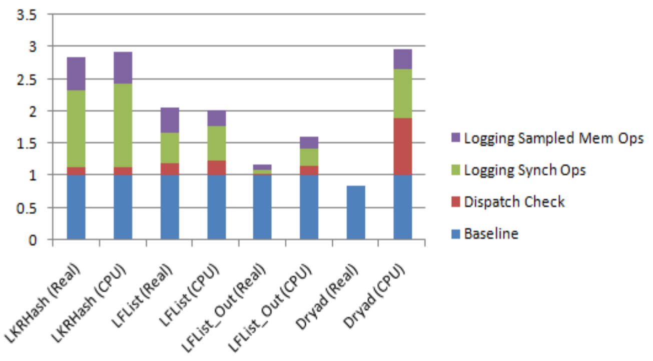

Figure 8. FeatherLite slowdown over the uninstrumented application.

6. Prior Work

In this section we discuss prior work in two areas related to this paper: data race detectors and samplers.

6.1 Data Race Detection

The two general approaches for data race detection can be classified into static and dynamic techniques. Static techniques use type-based analysis [5, 15, 33, 37] or model checkers [17, 34] or implement the lockset algorithm [14, 40]. There are techniques that statically implement a lockset [38] based algorithm [40, 14, 33]. Naik et al. [27] recently proposed an analysis method that consists of a set of techniques that are applied in series like reachability and alias analysis to reduce the number of false data races. Static techniques can be complete in that they can find all the potential data races in a program. But static techniques are not scalable to large code bases as they have exponential algorithmic complexity. Also, they tend to make conservative assumptions that lead them to report a large number of false data races. For example, in one of the very recent proposals [27], for one program `jdbm`, the static analysis returned 91 data races, but only 2 of them were found to sources of real bugs. This places a tremendous burden on the developer or tester to track down the true data races, and this wastes a lot of time, or the tool is not used, or bugs go unchecked. We focus on a dynamic analysis technique, since it can significantly reduce the number of candidate data races that need to be examined by the programmers.

Dynamic analysis techniques are either lockset based [38, 41, 29] or happens-before based [19, 28, 2, 7, 8, 10, 9, 39, 31, 35, 26] or a hybrid of the two [11, 42, 30, 32, 13]. Dynamic techniques are scalable to applications with large code bases and are also more precise than static tools as they analyze an actual execution of a program. The downside is that they have much less coverage of data races (false negatives), as they only examine the dynamic path of one execution the program. However, the number of false negatives can be reduced by increasing the number of tests.

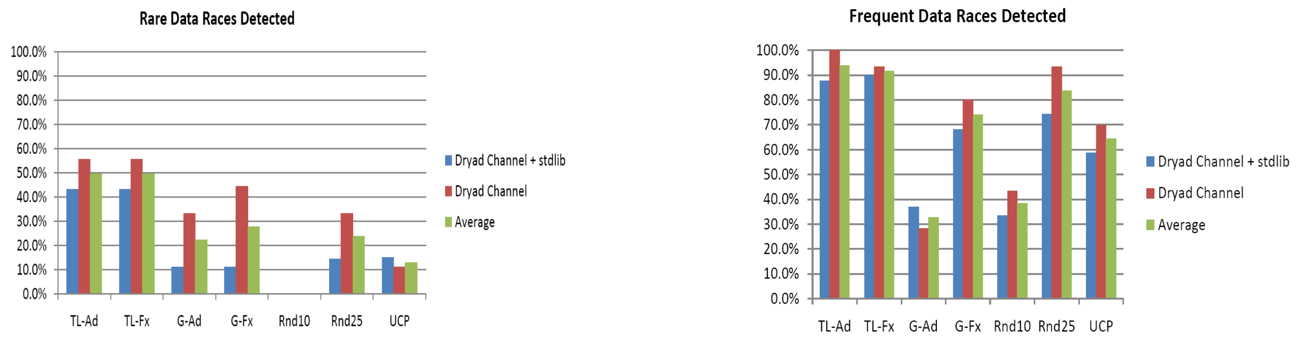

Figure 7. Samplers' data race detection rate for rare versus frequent races.

One of the main limitations of a dynamic data race detection tool has been its high run-time overhead, which perturbs the execution behavior of the application. Apart from consuming users time, a heavy-weight data race detector is not useful for finding bugs that would manifest in a realistic execution of an application. There has been attempts to ameliorate the performance cost of dynamic analysis using static optimizations for programs written in strongly typed languages [6]. Dynamic data race detectors for managed code [42] also has the advantage that the runtime system already incurs the cost of maintaining meta-data for the objects, which they make use of. For unmanaged code like C and C++, however, the runtime performance overhead of data race detection remains to be high. Intel's Thread-Checker [36], for example, incurs about 200x overhead to find data races. In this paper, we propose an efficient sampling mechanism that pays the cost for logging only a small fraction of the program execution, but is effective in detecting a majority of the data races. Unlike existing data race detectors, it also gives the user an ability to tradeoff performance cost with coverage (number of false negatives).

6.2 Sampling Techniques

Arnold et al. [3] proposed sampling techniques to reduce the overhead of instrumentation code in collecting profiles for feedback directed optimizations. Chilimbi and Hauswirth proposed an adaptive sampler for finding memory leaks [16]. We extend their solution for sampling multi-threaded programs, and show that samplers can be effectively used to find data races as well. Liblit et al. used statistical methods to collect samples from multiple production runs to isolate assertion and memory corruption errors [21, 22], but they did not study concurrency bugs.

7. Conclusion

Multi-threaded programs are hard to understand and debug. Dynamic data race detectors can automatically find concurrency bugs with a very high accuracy, which would be of immense help to the programmers. However, a significant impediment to their adoption is their runtime overhead. Programmers shy away from using heavy-weight dynamic tools as they cannot analyze a realistic execution of their application.

This paper argues for using sampling to ameliorate the runtime performance overhead of dynamic data race detectors. Our best sampler, thread local adaptive sampler, logs less than 5% of memory accesses but can detect more than 75% of data races.

While sampling might reduce coverage of these tools, it provides a knob, which the programmer can use to trade-off performance with coverage. Many tools never find acceptance in some of the product groups because of their high runtime overhead. With such a knob, programmers would be able to specify the performance penalty that they are willing to pay, and they would get a coverage that is commensurate with the performance penalty they paid.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer architecture*, 1991.

[3] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2001.

[4] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Programming Language Design and Implementation (PLDI)*, pages 68–78, 2008.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications*, 2002.

[6] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multi-threaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[7] J. D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.

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

[9] J. M. Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33, 1991.

[10] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, 1990.

[11] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, 1991.

[12] Joe Duffy. A query language for data parallel programming: invited talk. In *DAMP*, page 50, 2007.

[13] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–255, New York, NY, USA, 2007. ACM.

[14] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, 2003.

[15] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, 2000.

[16] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 156–164, New York, NY, USA, 2004. ACM.

[17] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, 2004.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, pages 59–72, 2007.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] Generic concurrent lock-free linked list — http://www.cs.rpi.edu/ bushl2/project_web/page5.html.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, New York, NY, USA, 2003. ACM.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM.

[23] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, 2005.

[24] Microsoft. Phoenix compiler. *http://research.microsoft.com/Phoenix/*.

[25] Microsoft. Thread execution blocks. *http://msdn.microsoft.com/en-us/library/ms686708.aspx*.

[26] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 235–244, 1991.

[27] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, 2006.

[28] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[29] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. *Third Virtual Machine Research & Technology Symposium*, pages 127–138, May 2004.

[30] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, 2003.

[31] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, pages 47–57, 1996.

[32] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, 2003.

[33] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, 2006.

[34] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, 2004.

[35] M. Ronsse and K. de Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Proceedings of Automated and Algorithmic Debugging*, Nov 2000.

[36] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 34–41, New York, NY, USA, 2006. ACM.

[37] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, 2005.

[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):391–411, 1997.

[39] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[40] N. Sterling. Warlock - a static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, 1993.

[41] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, 2001.

[42] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.