LiteRace: 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. They are precise in the sense that they only report actual data races. However, 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 LiteRace, 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 LiteRace using Microsoft's Phoenix compiler. Our experiments with several Microsoft programs, Apache, and Firefox show that LiteRace is able to find more than 70% of data races by sampling less than 2% of memory accesses in a given program execution.

Categories and Subject Descriptors D. Software [*D.2 Software Engineering*]: D.2.5 Testing and Debugging – Debugging aids

General Terms Algorithms, Experimentation, Reliability, Verification

Keywords Sampling, Dynamic Data Race Detection, Concurrency Bugs

1. Introduction

Multi-threaded programs are notoriously difficult to get right, largely due to the non-deterministic way in which threads in the program interleave during execution. 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 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 unexpected memory model effects of the underlying compiler [23, 6] 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 [7, 17, 33, 37, 19, 34, 16, 40, 27] provide maximum coverage by reasoning about data races on all execution paths. However, they tend to make conservative assumptions that lead to a large number of false data races. On the other hand, dynamic techniques [38, 30, 43, 15] are more precise than static tools, but their coverage is limited to the paths and thread interleavings explored at runtime. In practice, the coverage of dynamic tools can be increased by running more tests.

A severe limitation of dynamic data races detectors is their runtime overhead. Data race detectors like RaceTrack [43] that are implemented as part of a managed runtime system, incur about 2x to 3x slowdown. Data race detectors for unmanaged programs such as Intel's Thread Checker [36], incur performance overhead on the order of 200x. Such a large performance overhead prevents the wide-scale adoption of dynamic data-race detectors in practice. First, such a severe overhead dramatically reduces the amount of testing that can be done for a given amount of resources. More importantly, programmers and testers shy away from using intrusive tools that do not allow them to test realistic program executions.

The main reason for this very large performance overhead is that dynamic data-race detection requires analyzing every memory operation executed by the program. In this paper, we propose to use *sampling* to address this issue. 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, a sampling-based data-race detector may seem unlikely to succeed. 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 to 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 are likely to 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 either have already been found and fixed, or 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 rate, while the sampling rate for hot regions is adaptively reduced to a very small value. In this way, the adaptive sampler avoids slowing down the performance-critical hot regions of a program.

This paper describes an implementation of the proposed sampling algorithm in a tool called LiteRace, and demonstrates its effectiveness on a wide range of programs. LiteRace is implemented using the Phoenix compiler [24] to statically rewrite (x86) pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06. . . $5.00

gram binaries. For every function, LiteRace produces an instrumented copy of the function that logs all memory accesses and synchronization operations. In addition, LiteRace adds a check before every function entry. This dynamic check switches the execution between the uninstrumented and instrumented functions based on sampling information which is maintained for each thread. Our sampling technique is an extension of the adaptive profiling technique used in SWAT [18] for detecting memory leaks. The key difference is that our sampler needs to be "thread-aware". We want to avoid the situation where a code region that becomes hot due to repeated execution by a certain thread is not sampled when another concurrent thread executes it for the first time. To accomplish this, LiteRace maintains separate profiling information for each thread.To our knowledge, LiteRace is the first data-race detection tool that uses sampling to reduce the runtime performance cost.

This paper describes many of the challenges and trade-offs involved in building a tool like LiteRace. One of our key requirements for LiteRace is that it never report 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, although LiteRace samples memory accesses, it still captures *all* the synchronizations in the program. This is necessary to ensure that there are no false positives, as is explained in Section 3.2.

By sampling only a portion of the memory accesses, LiteRace is able to reduce the cost of logging meta-data about memory accesses, which can easily become a performance bottleneck in dynamic race detectors. The log of the sampled memory accesses and all of the synchronization operations can 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 [21] or a lockset based detector [38]. We chose to use happens-before based detection since it avoids reporting false races.

This papers makes the following contributions:

- We demonstrate that the technique of sampling can be used to significantly reduce the runtime overhead of a data race detector without introducing any additional false positives. LiteRace is the first data-race detection tool that uses sampling to reduce the runtime performance cost. As LiteRace permits users to adjust the sampling rate to provide 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. We show that a naive random sampler is inadequate for maintaining a high detection rate while using a low sampling rate. We propose a more effective adaptive sampler that heavily samples the first few executions of a function *in each thread*.
- We implemented LiteRace using the Phoenix compiler, and used it to analyze Microsoft programs such as ConcRT and Dryad, open-source applications such as Apache and Firefox, and two synchronization-heavy micro-benchmarks. The results show that by logging less than 2% of memory operations, we can detect more than 70% 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 detection, 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. We briefly discuss future work in Section 7 and conclude in Section 8.

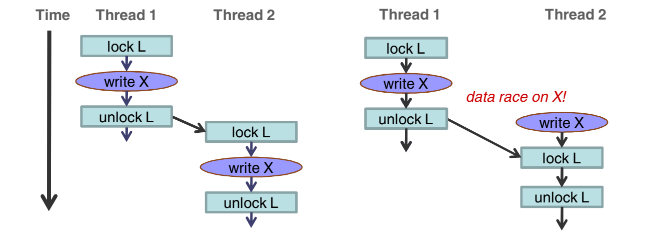


Figure 1. Examples of properly and *im*properly synchronized accesses to a memory location X. Edges between nodes represent a happens-before relationship. There is no data race for the example on the left, because there is a happens-before relation (due to unlock and lock operations) between the two writes to the location X. However, for the example on the right, there is no happens-before relation between the two writes. Thus, it has a data race.

2. Background

Dynamic data race detectors [38, 43] incur a high runtime overhead and we seek to address this problem 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 [21, 11] 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-based 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

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

(HB2) `a`$\longrightarrow$`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 `a`$\longrightarrow$`b` and `b`$\longrightarrow$`c`, then `a`$\longrightarrow$`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. In addition to being precise, another advantage that happens-before race detection has over the lockset-based approach is that it supports a wide range of synchronization paradigms, 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.

Figure 1 shows how the happens-before relationship is used to find data races. The edges between instructions indicate a happens-before relationship derived using rule HB1 or HB2. Transitively, by HB3, if there is a path between any two nodes, then there is a happens-before relationship between the two nodes. The example on the left in Figure 1 shows 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 the example shown on the right in Figure 1, thread 2 accesses a shared memory location without proper synchronization. Because there is no path between the two writes, the two writes are involved in a data race.

2.2 Sources 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 additional 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 [21, 28, 2, 9, 10, 12, 11, 39, 31, 35, 26] use vector clocks to keep track of the times of all the memory operations along with the address of the locations they accessed. Maintaining such meta-data further slows down the program execution due to increased memory cost.

3. LiteRace Overview

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

LiteRace has two key goals. First, LiteRace should not add too much runtime overhead during dynamic data-race detection. Our eventual goal is to run LiteRace 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, LiteRace should never report a *false* data race. Data races are very difficult to debug and triage. False positives severely limit the usability of a tool from a developer's perspective. This second goal has influenced many of our design decisions in LiteRace.

3.1 Case for Sampling

The key premise behind LiteRace is that sampling techniques can be effective for data-race detection. While a sampling approach has the advantage of reducing the runtime overhead, the main trade-off 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* data races in the program anyway. They can only find data races on thread interleavings and paths explored at runtime. Furthermore, a sampling-based detector, with its low overhead, would encourage users to widely deploy it on many more executions of the program, possibly achieving better coverage.

Another key advantage is that sampling techniques provide a useful *knob* that allow users to trade runtime overhead for coverage. For instance, users can increase the sampling rate for interactive applications that spend most of their time waiting for user inputs. In such cases, the overhead of data-race detection is likely to be masked by the I/O latency of the application.

3.2 Events to Sample

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

- Synchronization operations along with a logical timestamp that reflects the happens-before relation between these operations.

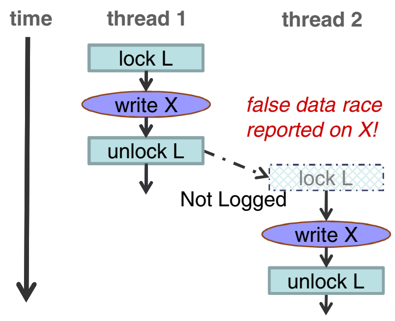


Figure 2. Failing to log a synchronization operation results in loss of happens-before edges. As a result, a false data race on X would be reported.

- Reads and writes to memory are logged in the program order, logically happening at the timestamp of the preceding synchronization operation of the same thread.

These logs can then be analyzed offline or during program execution (§4.4). The above information allows a data-race detector to construct the happens-before ordering between synchronization operations and the memory operations executed in different threads. A data race is detected if there is no synchronization ordering between two accesses to the same memory location, and at least one of them is a write.

Clearly instrumenting code to log every memory access would impose 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 fewer 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* the synchronization events in order to avoid reporting false data races. Figure 2 shows why this is the case. Synchronization operations induce happens-before orderings between program events. Any missed synchronization operation can result in missing edges in the happens-before graph. The data-race detection algorithm will therefore incorrectly report false 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, the number of synchronization operations is small compared to the number of instructions executed in a program. Thus, logging all synchronization operations 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, a good strategy for selecting which memory accesses to log is essential in order not to miss too many races. A data race involves two accesses and a sampler needs to successfully log both of them to detect the race. We describe a sampler that accomplishes this below.

3.3 Sampler Granularity

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 3. 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 3) 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 function call, we have to ensure that the dispatch code is as efficient as possible.

3.4 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. A naive random sampler does not meet these requirements as we show in Section 5.

Our sampler is an extension of the adaptive bursty sampler [18], previously shown to be successful for detecting memory leaks. An adaptive bursty sampler starts off by analyzing a code region at a 100% sampling rate, which means that the sampler always runs the instrumented copy of a code region the first time it is executed. Since the sampler is bursty, when it chooses to run the instrumented copy of a region, it does so for several consecutive executions. The sampler is adaptive in that after each bursty sample, a code region's sampling rate is decreased until it reaches a lower bound.

To make the adaptive bursty sampler effective for data-race detection, we extend the above algorithm by making it "thread local". The rationale is that, at least in reasonably well-tested programs, data races 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 it is likely to be a *benign* or intentional data race. In a "global" adaptive bursty sampler [18], a particular code region can be considered "hot" even when a thread executes it for the first time. This happens when other threads have already executed the region many times. We avoid this in LiteRace by maintaining separate sampling information for each thread, effectively creating a "thread local" adaptive bursty sampler. Our experiments (§5) show that this extension significantly improves the effectiveness of LiteRace.

Note that a thread-local adaptive sampler can also find some data races that occur between two hot regions or between a hot and a cold region. The reason is that our adaptive sampler initially assumes that all the regions are cold, and the initial sampling rate for every region is set to 100%. Also, the sampling rate for a region is never reduced below a lower bound. As a result, our sampler, even while operating at a lower sampling rate, might still be able to gather enough samples for a frequently executed hot region. Because of these two aspects of our adaptive sampler we find some, but not all, data races between hot-hot regions and hot-cold regions in a program.

4. LiteRace Implementation

This section describes the implementation details of LiteRace.

4.1 Instrumenting the Code

LiteRace is based on static instrumentation of x86 binaries and does not require the source code of the program. We use the Phoenix [24] compiler and analysis framework to parse the x86 executables and perform the transformation depicted in Figure 3. LiteRace creates two versions for each function: an *instrumented* version that logs all the memory operations and an *uninstrumented* version that does not log any memory operation. As explained in Section 3, avoiding false positives requires instrumenting *both* the instrumented and the uninstrumented versions to log synchronization operations. Then, LiteRace inserts a dispatch check at every function entry. This check decides which of the two versions to invoke for a particular call of the function at runtime.

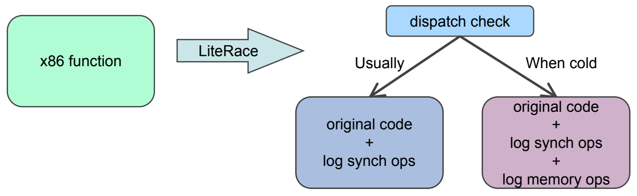

x86 function
LiteRace
dispatch check
Usually
When cold
original code + log synch ops
original code + log synch ops + log memory ops

Figure 3. LiteRace Instrumentation.

Synchronization Op	SyncVar	Add'l Sync?
Lock / Unlock	Lock Object Address	No
Wait / Notify	Event Handle	No
Fork / Join	Child Thread Id	No
Atomic Machine Ops	Target Memory Addr.	Yes

Table 1. Logging synchronization operations.

In contrast to prior adaptive sampling techniques [18], LiteRace maintains profiling information *per thread*. For each thread, LiteRace maintains a buffer in the thread-local storage that is allocated when the thread is created. This buffer contains two counters for each instrumented function: the frequency counter and the sampling counter. The frequency counter keeps track of the number of times the thread has executed a function and determines the sampling rate to be used for the function (a frequently executed function will be sampled at a lower sampling rate). The sampling counter is used to determine when to sample the function next. On function entry, the dispatch check decrements the sampling counter corresponding to that function. If the sampling counter's value is non-zero, which is the common case, the dispatch check invokes the uninstrumented version of the function. When the sampling counter reaches zero, the dispatch check invokes the instrumented version of the function, and sets the sampling counter to a new value based on the current sampling rate for the function as determined by the frequency counter.

As the dispatch check is executed on every function entry, it is important to keep the overhead of this check low. To avoid the overhead of calling standard APIs for accessing thread-local storage, LiteRace implements an inlined version 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 and restore 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.

For each synchronization operation, LiteRace logs a *SyncVar* that uniquely identifies the synchronization object and a logical timestamp that identifies the order in which threads perform operations on that object. Table 1 shows how LiteRace determines the SyncVar for various synchronization operations. For instance, LiteRace uses the address of the lock object as a SyncVar for `lock` and `unlock` operations. The logical timestamp in the log should ensure that if `a` and `b` are two operations on the same SyncVar and $a \longrightarrow b$ then `a` has a smaller timestamp than `b`. The simplest way to implement the timestamp is to maintain a global counter that is atomically incremented at every synchronization operation. However, the contention introduced by this global counter can dramatically slowdown the performance of LiteRace-instrumented programs on multi-processors. To alleviate this problem, we use one of 128 counters uniquely determined by a hash of the SyncVar for the logical timestamp.

To ensure the accuracy of the happens-before relation, it is important that LiteRace computes and logs the logical timestamp *atomically* with the synchronization operation performed. For some kinds of synchronization, we are able to leverage the semantics of the operation to guarantee this. 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. For wait/notify operations, LiteRace increments and logs the timestamp before the notify operation and after the wait operation to guarantee consistent ordering. A similar technique is used for fork/join operations.

For some synchronization operations, however, LiteRace is forced to add additional synchronization to guarantee atomic timestamping. 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 introduce a critical section to ensure that the compare-and-exchange and the logging and incrementing of the timestamp are all executed atomically. Without this, LiteRace can generate timestamps for these operations that are inconsistent with the actual order. Our experience shows that this additional effort is absolutely essential in practice and otherwise results in hundreds of false data races.

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 performed during a prior allocation. To avoid such false positives, LiteRace 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 LiteRace 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 of 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 logged events are processed using a standard implementation [36] of the happens-before based data-race detector described in Section 2.1. We did not use a lock-set based data-race detection algorithm to avoid false positives. However, the proposed sampling algorithms could be useful for lock-set based data-race detectors as well.

Benchmarks	Description	# Fns	Bin. Size
Dryad	Library for distributed data-parallel apps	4788	2.7 MB
ConcRT	.NET Concurrency runtime framework	1889	0.5 MB
Apache 2.2.11	Web server	2178	0.6 MB
Firefox 3.6a1pre	Web browser	8192	1.3 MB

Table 2. Benchmarks used. The number of functions and the binary size includes executable and any instrumented library files.

5. Results

In this section we present our experimental results. We begin by describing our benchmarks (§5.1) and the samplers that we evaluate (§5.2). In Section 5.3 we compare the effectiveness of various samplers in detecting data races. We show that our thread-local adaptive sampler achieves a high data-race detection rate, while maintaining a low sampling rate. Section 5.4 discusses the performance and log size overhead of thread-local adaptive sampler implemented in LiteRace, and compares it to an implementation that logs all the memory operations. All experiments were run on a Windows Server 2003 system with two dual-core AMD Opteron processors and 4 GB of RAM.

5.1 Benchmarks

We selected the four industrial-scale concurrent programs listed in Table 2 as our benchmarks. Dryad is a distributed execution engine, which allows programmers to use a computing cluster or a data center for running coarse-grained data-parallel applications [20]. The test harness we used for Dryad was provided by its lead developer. The test exercises the shared-memory channel library used for communication between the computing nodes in Dryad. We experimented with two versions of Dryad, one with the standard C library statically linked in (referred to as Dryad-stdlib), and the other without. For the former, LiteRace instruments all the standard library functions called by Dryad. Our second benchmark, ConcRT, is a concurrent run-time library that provides lightweight tasks and synchronization primitives for developing data-parallel applications. It is part of the parallel extensions to the .NET framework [14]. We used two different test inputs for ConcRT: Messaging, and Explicit Scheduling. These are part of the ConcRT concurrency test suite. We use Apache, an open-source HTTP web server, as our third benchmark. We evaluate the overhead and effectiveness of LiteRace over two different inputs for Apache (referred to as Apache-1 and Apache-2). The first consists of a mixed workload of 3000 requests for a small static web page, 3000 requests for a larger web page, and 1000 CGI requests. The second consists solely of 10,000 requests for a small static web page. For both workloads, up to 30 concurrent client connections are generated by Apache's benchmarking tool. Our final benchmark is Firefox, the popular open-source web browser. We measure the overhead and sampler effectiveness for the initial browser start-up (Firefox-Start) and for rendering an html page consisting of 2500 positioned DIVs (Firefox-Render).

5.2 Evaluated Samplers

The samplers that we evaluate are listed in Table 3. The "Short Name" column shows the abbreviation we will use for the samplers in the figures throughout the rest of this section. The table also shows the effective sampling rate for each sampler. The effective sampling rate is the percentage of memory operations that are logged by a sampler. Two averages for effective sampling rate are

Sampling Strategy	**Short Name**	**Description**	**Weighted Average ESR**	**Average ESR**
Thread-local Adaptive	TL-Ad	Adaptive back-off per function / per thread (100%,10%,1%,0.1%); bursty	1.8%	8.2%
Thread-local Fixed 5%	TL-Fx	Fixed 5% per function / per thread; bursty	5.2%	11.5%
Global Adaptive	G-Ad	Adaptive back-off per function globally (100%, 50%, 25%, ... , 0.1%); bursty	1.3%	2.9%
Global Fixed	G-Fx	Fixed 10% per function globally; bursty	10.0%	10.3%
Random 10%	Rnd10	Random 10% of dynamic calls chosen for sampling	9.9%	9.6%
Random 25%	Rnd25	Random 25% of dynamic calls chosen for sampling	24.8%	24.0%
Un-Cold Region	UCP	First 10 calls per function / per thread are NOT sampled, all remaining calls are sampled	98.9%	92.3%

Table 3. Samplers evaluated along with their short names used in figures, short descriptions, and effective sampling rates averaged over the benchmarks studied. The weighted average uses the number of memory accesses in each benchmark application as a weight.

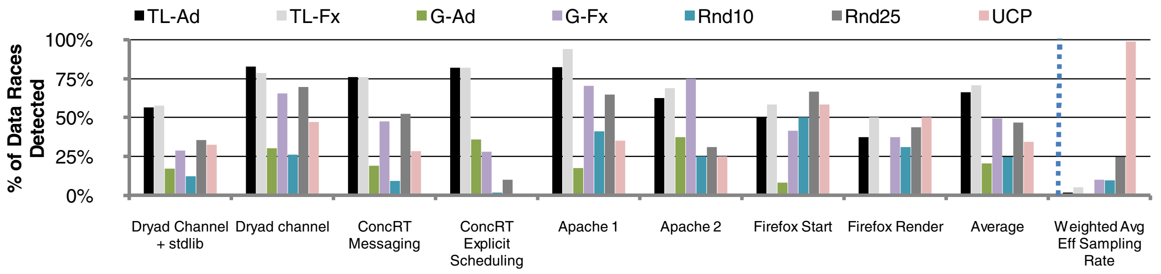
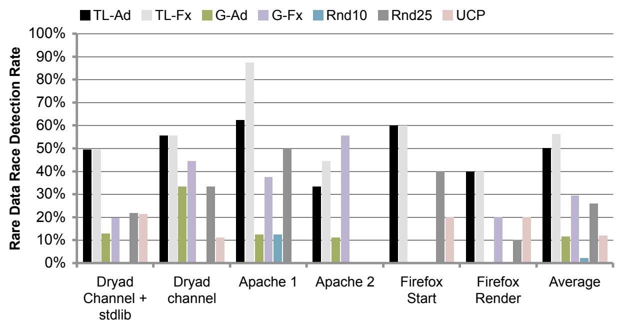


Figure 4. Proportion of static data races found by various samplers. The figure also shows the weighted average effective sampling rate for each sampler, which is the percentage of memory operations logged (averaged over all the benchmarks).



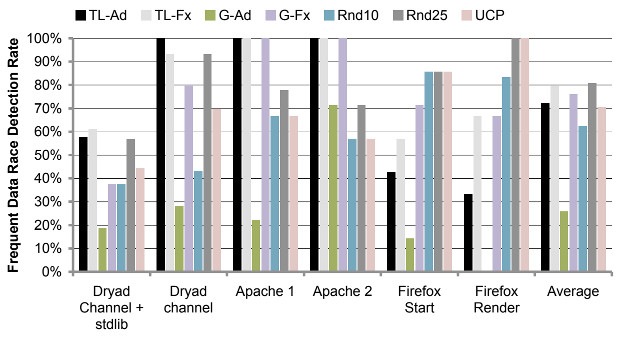


Figure 5. Various samplers' detection rate for rare (on the left) and frequent (on the right) static data races.

shown. One is just the average of the effective sampling rates over the nine benchmark-input pairs described in Section 5.1. The other is the weighted average, where the weight for a benchmark-input pair is based on the number of memory operations executed at runtime.

LiteRace's thread-local adaptive sampler is the first one listed in the table. For each thread and for each function, this sampler starts with a 100% sampling rate and then progressively reduces the sampling rate until it reaches a base sampling rate of 0.1%. To understand the utility of this adaptive back-off, we evaluate a thread-local fixed sampler, which uses a fixed 5% sampling rate per function per thread. The next two samplers are "global" versions of the two samplers that we just described. The adaptive back-off for the "global" sampler is based on the number of executions of a function, irrespective of the calling thread. This global adaptive sampler is similar to the one used in SWAT [18], except that we use a higher sampling rate. Even with a higher rate, our experiments show that the global samplers are not as effective as the thread-local samplers in finding data races. The four samplers mentioned thus far are "bursty". That is, when they decide to sample a function, they do so for ten consecutive executions of that function. The next two samplers are based on random sampling and are not bursty. Each function call is randomly sampled based on the chosen sampling rate (10% and 25%). The final sampler evaluates our cold-region hypothesis by logging only the "uncold" regions. That is, it logs all but the first ten calls of a function per thread.

5.3 Effectiveness of Samplers Comparison

In this section, we compare different samplers and show that the thread-local adaptive sampler is the most effective of all the samplers we evaluated. For our evaluation, we group each data race detected by our tool based on the pair of instructions (identified by the value of the program counter) that participate in the data race. We call each group a *static* data-race. From the user's perspective, a static data-race roughly corresponds to a possible synchronization error in the program. Table 4 shows the number of static data-races that LiteRace finds for each benchmark-input pair. The table also distinguishes between rare and frequent static data-races, based on the number of times a particular static data-race manifests at runtime.

To have a fair comparison, different samplers need to be evaluated on the *same* thread interleaving of a program. However, two different executions of a multi-threaded program are not guaranteed to yield the same interleaving even if the input is the same. To compare the effectiveness of the various samplers in detecting data races accurately, we created a modified version of LiteRace that performs full logging, where all synchronization and all memory operations are logged. In addition to full logging, upon function entry, we execute the "dispatch check" logic for 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 data-race detection on the complete log, we find all the data races that happened during the program's execution. We can then perform data-race detection on the subset of the memory operations that a particular sampler would have logged. Then, by comparing the results with those from the complete log, we are able to calculate the *detection rate*, which is the proportion of data races detected by each of the samplers. When we analyze the performance and space overhead in Section 5.4, however, we experiment with only LiteRace's thread-local adaptive sampler turned on.

Each application was instrumented using our modified version of LiteRace described above. We ran the instrumented application three times for each benchmark. The detection rate we report for each benchmark is the average of the three runs. The results for overall data-race detection rate are shown in Figure 4. The results are grouped by benchmarks with a bar for each sampler within each group. The weighted average effective sampling rate for each of the samplers (discussed in Section 5.2) is also shown as the last group. A sampler is effective if it has a very low effective sampling rate along with a high data-race detection rate. Notice that the proposed LiteRace sampler (TL-Ad) achieves this, as it detects about 70% of all data-races by sampling only 1.8% of all memory operations. The non-adaptive fixed rate thread-local sampler also detects about 72% of data-races, but its effective sampling rate is 5.2% (more than 2.5x higher than the TL-Ad sampler). Clearly, among the thread-local samplers, the adaptive sampler is better than the fixed rate sampler.

The two thread-local samplers outperform the two global samplers. Though the global adaptive sampler logs only 1.3% of memory operations (comparable to our thread-local adaptive sampler), it detects only about 22.7% of all data-races (about 3x worse than TL-Ad). The global fixed rate sampler logs 10% of memory operations, and still detects only 48% of all data-races.

All the four samplers based on cold-region hypothesis are better than the two random samplers. For instance, a random sampler finds only 24% of data-races, but logs 9.9% of all memory operations.

Another notable result from the figure is that of the "Un-Cold Region" sampler, which logs all the memory operations except those executed in the cold-regions (§5.2). It detects only 32% of all data-races, but logs nearly 99% of all memory operations. This result validates our cold-region hypothesis.

5.3.1 Rare Versus Frequent Data Race Detection

We have so far demonstrated that a thread-local adaptive sampler finds about 70% of all static data-races. If a static data-race occurs frequently during an execution, then it is likely that many sampling strategies would find it. It is more challenging to find data races that occur rarely at run-time. To quantify this, we classified all of the static data races that were detected (using the full, unsampled log) based on the number of times that a static data-race occurs in an execution. We classified as rare those racing instruction pairs that occurred fewer than 3 times for each million non-stack memory instructions executed. The rest are considered frequent. The number of rare and frequent data races for each benchmark-input pair is shown in Table 4 (some of the data races found could be benign). The various samplers' data-race detection rates for these two categories are shown in Figure 5.

Benchmarks	# races found	# Rare	# Freq
Dryad Channel + stdlib	19	17	2
Dryad Channel	8	3	5
Apache-1	17	8	9
Apache-2	16	9	7
Firefox Start	12	5	7
Firefox Render	16	10	6

Table 4. Number of static data-races found for each benchmark-input pair (median over three dynamic executions), while logging all the memory operations. These static data-races are classified into rare and frequent categories. A static data-race is rare, if it is detected less than 3 times per million non-stack memory instructions during any execution of the program.

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.

5.4 Analysis of Overhead

In Section 5.3 we presented results showing that the thread-local adaptive sampler performs well in detecting data-races for a low sampling rate. Here we present the performance and log size overhead of thread-local adaptive sampler implemented in LiteRace. We show that, on average, it incurs about 28% performance overhead for our benchmarks when compared to no logging, and is up to 25 times faster than an implementation that logs all the memory operations.

Apart from the benchmarks used in Section 5.3, we used two additional compute and synchronization intensive micro-benchmarks for our performance study. LKRHash is an efficient hash table implementation that uses a combination of lock-free techniques and high-level synchronizations. LFList is an implementation of a lock-free linked list available from [22]. LKRHash and LFList execute synchronization operations more frequently than the other real world benchmarks we studied. These micro-benchmarks are intended to test LiteRace's performance in the adverse circumstance of having to log many synchronization operations.

To measure the performance overhead, we ran each of the benchmarks ten times for each of four different configurations. The first configuration is the baseline, uninstrumented application. Each of the remaining three configurations adds a different portion of LiteRace's instrumentation overhead: the first adds just the dispatch check, the second adds the logging of synchronization operations, and the final configuration is the complete LiteRace instrumentation including the logging of the sampled memory operations. By running the benchmarks in all of these configurations we were able to measure the overhead of the different components in LiteRace.

Figure 6 shows the cost of using LiteRace on the various benchmarks and micro-benchmarks. The bottom portion of each vertical bar in Figure 6 represents the time it takes to run the uninstrumented application (baseline). The overhead incurred by the various components of LiteRace are stacked on top of that. As expected, the synchronization intensive micro-benchmarks exhibit

Benchmarks	Baseline Exec Time	LiteRace Slowdown	Full Logging Slowdown	LiteRace Log Size (MB/s)	Full Logging Log Size (MB/s)
LKRHash	3.3s	2.4x	14.7x	154.5	1936.3
LFList	1.7s	2.1x	16.1x	92.5	751.7
Dryad+stdlib	6.7s	1x	1.8x	1.2	12.8
Dryad	6.6s	1x	1.14x	1.1	2.6
ConcRT Messaging	9.3s	1.03x	1.08x	0.7	10.6
ConcRT Explicit Scheduling	11.5s	2.4x	9.1x	4.6	109.7
Apache-1	17.0s	1.02x	1.4x	1.2	41.9
Apache-2	3.0s	1.04x	3.2x	4.0	260.7
Firefox Start	1.8s	1.44x	8.89x	7.4	107.0
Firefox Render	0.61s	1.3x	33.5x	19.8	731.1
Average	6.15s	1.47x	9.09x	28.6	396.5
Average (w/o Microbench)	7.06s	1.28x	7.51x	5.0	159.6

Table 5. Performance overhead of LiteRace's thread-local adaptive sampler and full logging implementation when compared to the execution time of the uninstrumented application. Log size overhead in terms of MB/s is also shown.

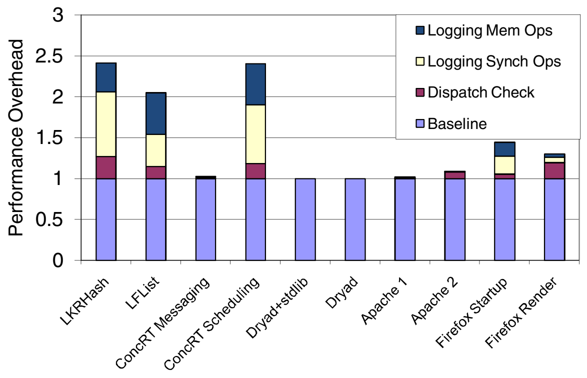


Figure 6. LiteRace slowdown over the uninstrumented application.

the highest overhead, between 2x and 2.5x, since we must log all synchronization operations to avoid false positives. The ConcRT Scheduling test also has a high proportion of synchronization operations and exhibits overhead similar to the micro-benchmarks. The more realistic application benchmarks show modest performance overhead of 0% for Dryad, 2% to 4% for Apache, and 30% to 44% for Firefox.

In order to evaluate the importance of sampling memory operations in order to achieve low overhead, we measured the performance of logging *all* the synchronization and memory accesses. Unlike the LiteRace implementation, this full-logging implementation did not have the overhead for any dispatch checks or cloned code. Table 5 compares the slowdown caused by LiteRace to the slowdown caused by full logging. The sizes of the log files generated for these two implementations are also shown in terms of MB/s. LiteRace performs better than full logging in all cases. The performance overhead over baseline when averaged over realistic benchmarks is 28% for the LiteRace implementation, while the full logging implementation incurs about 7.5x performance overhead.

The generated logs, as expected, are also much smaller in LiteRace. On average, LiteRace generated logs at the rate of 5.0 MB/s, whereas a full logging implementation generated about 159.6 MB/s.

6. Related 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

Prior data race detection can be broadly classified into static and dynamic techniques. Static techniques include those that use type-based analysis [7, 17, 33, 37] or data-flow analysis [40, 16, 27, 42] to ensure that all data accesses are consistently protected by locks. Many of these techniques are scalable and most are complete in that they find *all* data races in a program. The downside is that static techniques are inherently imprecise and typically report a large number of false data races that place a tremendous burden on the user of the tool. More importantly, these techniques are not able to handle synchronizations other than locks, such as events, semaphores, and I/O completion ports common in many systems programs. Thus, data accesses that are synchronized through these mechanisms will be falsely reported as potential data races. Model checking techniques [19, 34] are capable of handling such synchronizations, but are not scalable due to the complexity of their analysis. A dynamic tool, such as LiteRace, does not suffer these problems.

Dynamic analysis techniques are either lockset based [38, 41, 29] or happens-before based [21, 28, 2, 9, 10, 12, 11, 39, 31, 35, 26] or a hybrid of the two [13, 43, 30, 32, 15]. 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.

One of the main limitations of a dynamic data race detection tool is 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 have been attempts to ameliorate the performance cost of dynamic analysis using static optimizations for programs written in strongly typed languages [8]. Dynamic data race detectors for managed code [43] also have 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 high. Intel's ThreadChecker [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 for Dynamic Analysis

Arnold et al. [4] 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 [18]. We extend their solution to the sampling of multi-threaded programs, and show that samplers can be effectively used to find data races as well. QVM [3] is an extension to Java Virtual Machine that provides an interface to enable dynamic checking such as heap properties, local assertions, and typestate properties. It uses sampling to tradeoff accuracy with runtime overhead. The sampling technique used in QVM is object-centric, in that, all the events to a sampled object's instance are profiled. In contrast, our samplers are based on cold-region hypothesis.

7. Future Work

Our current LiteRace implementation samples code regions at the granularity of functions (Section 3.3). While this approach works very well for server applications like Apache, web browsers like Firefox, and highly concurrent programs like Dryad and ConcRT, it may not be the best possible implementation for computationally intensive scientific applications like Parsec [5]. These application often have loops with high trip count. Therefore sampling at a loop-level granularity might help improve the efficiency of LiteRace for these applications. Offline profiling can be used to identify loops with high trip count, which can then be instrumented to adaptively reduce the sampling rate of the loop within a single function execution.

8. Conclusions

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 programmers. However, a significant impediment to their adoption is their runtime overhead. Programmers shy away from heavy-weight dynamic tools that prevent them from testing realistic executions of their application. Moreover, the high overhead of such tools dramatically reduces the amount of testing possible for a given amount of computing and time resources.

This paper argues for sampling-based techniques to ameliorate the runtime performance overhead of dynamic data race detectors. We demonstrate that intelligent sampling can be effective in finding data races with acceptable runtime overhead. Our best sampler, the thread local adaptive sampler, logs less than 2% of memory accesses but can detect more than 70% of data races.

Another key advantage of a sampling-based technique is that it provides a knob in the form of sampling rate, which the programmer can use to trade-off performance for data-race coverage. Many testing tools never find acceptance in development teams 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 coverage that is commensurate with this penalty.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on this paper. We would also like to thank Trishul Chilimbi for helpful discussions.

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] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008.

[4] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, 2001.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[6] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, 2008.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, 2002.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, pages 59–72, 2007.

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

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

[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 '96: Operating System Design and Implementation*, 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, 2006.

[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] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, 2007.

[43] 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.