Automatically Classifying Benign and Harmful Data Races Using Replay Analysis

Satish Narayanasamy[†‡], Zhenghao Wang[†], Jordan Tigani[†], Andrew Edwards[†], Brad Calder[†‡]

[†]Microsoft
[‡]University of California, San Diego

Abstract

Many concurrency bugs in multi-threaded programs are due to data races. There have been many efforts to develop static and dynamic mechanisms to automatically find the data races. Most of the prior work has focused on finding the data races and eliminating the false positives.

In this paper, we instead focus on a dynamic analysis technique to automatically classify the data races into two categories – the data races that are potentially benign and the data races that are potentially harmful. A harmful data race is a real bug that needs to be fixed. This classification is needed to focus the triaging effort on those data races that are potentially harmful. Without prioritizing the data races we have found that there are too many data races to triage. Our second focus is to automatically provide to the developer a reproducible scenario of the data race, which allows the developer to understand the different effects of a harmful data race on a program's execution.

To achieve the above, we record a multi-threaded program's execution in a replay log. The replay log is used to replay the multi-threaded program, and during replay we find the data races using a happens-before based algorithm. To automatically classify if a data race that we find is potentially benign or potentially harmful, we replay the execution twice for a given data race – one for each possible order between the conflicting memory operations. If the two replays for the two orders produce the same result, then we classify the data race to be potentially benign. We discuss our experiences in using our replay based dynamic data race checker on several Microsoft applications.

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 Benign Data Races, Replay, Concurrency Bugs

1. Introduction

Automatically detecting data races is a very hard problem. Data race detection tools, even the dynamic analysis tools, tend to report a large number of data races. But only a handful of them are harmful. A *harmful* data race is one that is a source of a concurrency bug, which can affect the correctness of a program's execution. A developer will consider fixing only the harmful data races. Ideally, an automatic race detection tool should report only the harmful data races to a developer. However, many existing tools report data races that can never occur at all. Such data races are the *false positives*. Even if we manage to eliminate all the false positives, not all of the remaining true data races are harmful. In fact, only a small fraction of the true data races are actually harmful. In our work, we found that only 10% of the true data races are harmful. The remaining 90% were all *benign* data races - the data races that do not compromise program's correctness. Thus, the set of true data races is still too large for developers to manually examine and triage.

The first goal of our paper is to find and report only the potentially harmful data races, in order to improve programmer productivity. To achieve this, we propose a mechanism to automatically classify the *true data races* into *potentially benign* and *potentially harmful*. This allows the developers to prioritize the data races that need be triaged. We find that reporting accurate information about the potentially harmful data races is very important because triaging a data race bug is a tedious exercise. Triaging a data race bug is difficult for the following reasons:

- **Requires Domain Expertise:** Effects of a data race are hard to understand, as it involves analyzing multiple program states across multiple threads. Domain expertise is usually required to understand if a data race is benign or harmful in our production code.
- **Time Consuming:** The number of true data races is too large for the developers to go through and triage them all. It can be extremely time consuming for a developer to triage a data race, and those with the domain expertise are often very busy.
- **Hard to Figure Out:** Even if someone with domain expertise examines the data race, they tend to incorrectly believe it is benign when it is actually harmful, or vice versa.

Besides finding the harmful data races accurately, another problem is generating information that will help convince the developer about the existence of a data race bug. Therefore, our second goal is to generate a concrete reproducible scenario for a potentially harmful data race. The reproducible scenario helps the programmer in debugging and understanding the harmful effects of the data race reported.

In this paper, we take important steps towards meeting both of the above two goals. We have developed a dynamic data race detection tool that can automatically classify the data races into potentially benign and potentially harmful categories. An integral part of our solution is the ability to record a program's execution in a replay log and replay the program's execution using the log. The proposed dynamic data race detection analysis is performed offline, during replay. In addition to finding the potentially harmful

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'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006…$5.00

data races, the analysis also produces useful information for each data race. Using that information, a developer can replay a program execution in two different ways – the original execution order and an alternative order, which has the order of the two memory operations involved in the data race reversed. The two replays help the developer understand how a potentially harmful data race can produce different results based on the different interleavings between the two racing memory operations.

Data Race Detection: We focus on building a race detection tool with no false positives. Our data race detection algorithm is based on the *happens-before* relationship [17]. We determine that there is a data race between two memory operations executed in two different threads if (a) at least one of them is a write, and (b) there is no synchronization operation executed between the two memory operations (that is, there is no happens-before relation to provide an order for the two operations). Going by this strict definition of what a data race is, a happens-before based algorithm does not report any false positives. However, a happens-before algorithm still reports a large number of data races, out of which many are benign. In order for our tool to be used in practice, we need to prioritize the data races so that a developer can focus on fixing and understanding the potentially harmful data race bugs.

Data Race Classification: Our approach automatically classifies data races by leveraging the ability to replay the program's execution. The key concept behind our analysis is as follows. For a data race, the checker analysis tool replays the execution twice for the two different orders between the memory operations involved in the data race. If the two replays produce the same result, then the checker determines that the data race is potentially benign. Otherwise, it classifies the data race as potentially harmful. The data races that our tool marks as potentially benign are not examined by the developers, but only those marked as potentially harmful are examined. We keep track of the results of this analysis for each pair of memory operations involved in a data race. There can be many instances of the same data race during a program's execution and across several different executions. By analyzing these instances we can observe several effects of the data race. Thereby, we get a much clearer picture about how to classify the data race.

Data Race Report: At the end of our analysis, we provide the developer with precise information about the effects of each potentially harmful data race. The information includes the replay log and the two memory orders that were analyzed by the checker for the data race. One of the replays will produce the correct result and the other will produce a different result. Thus, a developer has the precise information about the memory operations involved in the data race, *and* also has the ability to replay the program in two different ways (two ways are the original execution order and the alternative order that is possible due to the data race) and understand the effects of the data race. If the same data race had occurred multiple times within the same or different execution scenario, we provide information for all of those instances to help the developer understand the various possible effects of a particular data race.

iDNA and Usage Model: We perform our dynamic data race analysis using the reply tool called iDNA developed by Bhansali et al. [3]. iDNA provides the ability to record a program's execution in a replay log. Using the replay log, iDNA can replay a multi-threaded program's execution, even in the presence of all forms of non-determinism, including system interactions (system calls, interrupts, DMAs etc.) and multi-processor interactions. We extend iDNA to provide the ability to replay with two different thread interleavings for a data race, and provide the ability to examine the results of both orderings to see if they result in the same execution. In using our approach in a development environment, iDNA is first used to gather the replay logs for the product's test scenarios, with an overhead of about 10x on average [3]. We then run our off-line replay analysis to find all of the data races, and the off-line analysis classifies the data races into potentially benign or potentially harmful. For the potentially harmful ones, we provide at least two replay scenarios that will show how the data race can result in two different outcomes. This information coupled with the ability to do reverse execution (also called time travel debugging) using iDNA [3] for the replays, provides a powerful platform for the developers to examine the potentially harmful data races.

In striving to achieve the above two goals, our approach can have incorrect classifications. We may classify some data races as potentially benign when they are harmful or vice versa. If we classify a benign data race as potentially harmful, then we end up using precious developer's time. But once those races are manually identified as benign, they are marked as benign to prevent them from being classified as potentially harmful in the future analysis. If we classify a harmful data race as benign, they will not be examined by the developer. However, later on, when analyzing a different test case, the analysis may find an instance of the data race that exposes it as potentially harmful. The data race will then be re-classified and reported to the developer. Thus, the more the number of test cases analyzed, the more likely harmful data races will be discovered (which is true for any dynamic analysis tool). This is a trade-off between coverage and accuracy that we make during development, because there are too many true data races found, and most of them are benign.

To summarize, this paper makes the following contributions:

- We present a happens-before based dynamic analysis algorithm to find a set of data races using the iDNA replay framework. The happens-before based analysis does not report false positives. However, the analysis still reports a lot of data races, most of which are benign.
- To our knowledge, we are the first to focus on the problem of automatically classifying the *true* data races into potentially benign and potentially harmful. This classification directs the focus of the developer on triaging the potentially harmful data races. The key idea is that for a data race to be harmful, the two different possible memory orders for the data race should produce different results. We describe how we extended the iDNA framework to replay the alternative memory orderings, and to determine if they result in the same execution or not. Then for each potentially harmful data race, we provide a replay scenario that allows the developer to understand the effects of the data race.
- We discuss our experiences in using our dynamic race classification approach on an extensively stress-tested build of Microsoft's Windows Vista and Internet Explorer. Our proposed technique was able to automatically filter out over half of the real benign data races, classifying them as potentially benign, which can be ignored by the developers. In addition all of the harmful data races were correctly classified as potentially harmful, which were reported to the developers and they all have been fixed in the production code.

2. Prior Work

The focus of this paper is to accurately classify data races in parallel applications based on the shared memory programming model. Prior work on data race detection can be broken into solutions based on static analysis and solutions based on dynamic analysis.

2.1 Static Analysis

Data races can be found using type-based static analysis techniques [4, 14]. A type-based technique requires the programmer

to specify the type of the synchronization operations [4, 14]. Automatically inferring information about the synchronization operations is difficult and there are some techniques that address this problem [32]. Static analysis can also be done using model checking techniques like BLAST [15] and KISS [28]. Model checking techniques can handle various synchronization idioms and can also produce counterexamples. The limitation of the model checking techniques is that the analysis algorithm does not scale well for large programs, which can limit their use.

There are techniques that statically implement a lockset [33] based algorithm [35, 13, 26]. Naik et al. [20] 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.

The primary limitation of the static analysis techniques is their accuracy in terms of the number of false positives reported. Also, an even bigger problem (which is true even for existing dynamic analysis techniques) is that, among the true data races reported, a large proportion of them are benign data races. Benign data races are very hard to distinguish from the harmful data races during static analysis. For example, in one of the very recent proposals [20], for one program `jdbm`, the analysis returned 91 data races (not false positives but real data races), but only 2 of them were found to be harmful. Static analysis techniques address this problem with manual annotations, but they require the programmer to get the annotation right, and there is a significant amount of existing code existing without annotations.

We focus on a dynamic analysis technique, since it can significantly reduce the number of candidate data races that need to be examined. The trade-off of course is that, the coverage will be lower than the static techniques. Also, we can generate different replay scenarios for a data race found during a dynamic analysis, which the user can use to understand the possible effects of the data race on a program's execution.

2.2 Dynamic Analysis

Dynamic analysis can be done either on-line or off-line. We first examine the trade-offs between the two approaches. Then we describe the dynamic race detection techniques in more detail and place our work in context.

2.2.1 When the Analysis is Performed: On-line Versus Off-line Analysis

A program's execution can be analyzed *on-line* when the program is executing to detect the data races. This approach incurs runtime overhead, and hence the dynamic analysis needs to be efficient in terms of performance. A majority of prior dynamic race detection techniques have focused on detecting data races on-line, either with instrumentation support [33] and with hardware support [1, 29]. There have also been attempts to ameliorate the performance cost of dynamic analysis using static optimizations [11, 38, 23, 25].

Alternatively, if we can efficiently record sufficient information about a program's execution to allow us to deterministically replay the execution, then we can do *off-line* analysis. The advantage of off-line analysis over on-line analysis is that the analysis itself does not have to be as performance efficient as it is has to be for on-line analysis. Only the recording part needs to be efficient. We can perform (many) sophisticated time consuming dynamic analysis over a recorded program's execution off-line. Also, the result of the analysis can enable the developer to examine the source of the data race by replaying the program.

There have been a few techniques that looked at doing off-line analysis [7, 30] to detect data races. However, both RaceFrontier [7] and RecPlay [30] do not attempt to record the non-deterministic interactions between the threads. As a result, they are limited in their analysis in that they are able to detect only the first data race in the recorded program execution. In contrast, we use the iDNA [3] infrastructure, which enables us to replay multi-threaded programs across all forms of non-determinism, including non-deterministic shared memory multiprocessor interactions. This allows us to examine all the data races in the recorded program execution.

2.2.2 How the Analysis is Performed: Happens-Before Versus Lockset

Dynamic race detection algorithms can be broadly classified into happens-before based algorithms [17, 21, 1, 6, 10, 8, 34, 24, 31, 19], lockset based algorithms [33, 36, 22, 2] and hybrid algorithms that combine the two [11, 38, 23, 25].

One class of data race detectors use the lockset algorithm. The lockset algorithm checks whether each shared variable in a program is consistently guarded by at least one lock. Eraser [33] implements the lockset algorithm using instrumentation to dynamically find the data races during a program's execution. This algorithm has been extended to object-oriented languages [25] and improved for precision and performance [2, 22, 36, 5]. The lockset algorithm is essentially a heuristics based algorithm and hence reports data races that can never occur at all (that is, it can report false positives). A recent work [18] reports that a lockset algorithm resulted in thousands of false positives for scientific applications.

There are race detectors that use the happens-before algorithm. The happens-before algorithm checks whether conflicting accesses to shared variables in a program are ordered by an explicit synchronization operation or not. Many dynamic race detectors implement the happens-before algorithm in software [31] . Hardware [19, 27] and Distributed-Shared-Memory [24, 29] implementations were also proposed to reduce the runtime overhead of these detectors. A recent hardware based proposal called ReEnact [27] detects data races using happens-before relation on-the-fly. Upon detection of a data race, it can rollback to a previous checkpoint and replay the execution. During replay, it tries to avoid the data race detected in the previous execution. The advantage of using a happens-before algorithm is that it can detect the data races with no false positives because the analysis is based on whether there are two unordered conflicting memory operations or not. However, the resulting coverage can be less than the lockset algorithm.

It is also possible to combine these two algorithms [11, 38, 23, 25] to get coverage close to a lockset algorithm, and at the same time reduce false positives using happens-before relations.

These prior dynamic data race detectors have not yet focused on classifying real data races as potentially benign versus harmful data races. For example, RaceTrack [38] found 48 warnings in CLR regression test suite out of which there were 8 false positives. But more importantly, 32 were benign data races and only 8 were found to be harmful during manual inspection. Distinguishing between the benign and the harmful data races is a hard problem. To our knowledge no prior work has attempted to automatically identify the potentially benign data races, which is the focus of our work. If can do that, then we can direct the developers's effort towards triaging the potentially harmful data races.

The focus in building our tool was to provide as much accurate information as possible. That is why we chose to use a happens-before based data race detection algorithm, since it does not report any false positive. Neverthless, our analysis can also be used for analyzing the data races reported by a lockset based algorithm and its variations. The analysis should be able to filter out the benign data races and also the false positives produced by those algorithms.

2.3 Atomicity Violation Detection

There have also been work on finding concurrency bugs by checking for atomicity violations [2, 12, 37, 18]. If we can know which

regions of code need to be executed atomically, then we can verify the atomicity properties either statically [2] or dynamically [12, 37, 18]. Also, there has been work on inferring the set of locks that need to be acquired to enforce the atomicity specified by the programmer [16]. Any violation of atomicity is a source of a bug, but every data race is not necessarily harmful. So checking for atomicity violations is more effective than finding data races. However, determining the atomic regions in itself is a significant challenge. Many techniques require the programmer to explicitly specify the atomic regions through annotations [2, 12, 16]. In SVD [37] and AVIO [18], the authors used heuristics to infer the atomic regions automatically. These methods are heuristic based, and as a result they report a high number of false positives when a code region is incorrectly determined to be atomic.

3. Finding Happens-before Replay Data Races

In this section, we provide a brief overview of the iDNA [3] record and replay mechanism. We then discuss a happens-before based data race detection algorithm that we implemented by extending the iDNA replayer. Our happens-before based data race detector does not report false positives. In other words, if our detector finds a data race in a program's execution, then it guarantees that there is at least one instance of the data race in the execution.

3.1 iDNA Recorder

iDNA [3] provides the ability to record a multi-threaded program's execution in a replay log, which can be used to replay the execution. Here we will briefly discuss how iDNA works, but more details can be found in [3].

iDNA uses a load-based checkpointing scheme to record a program's execution. Let us first just consider a single threaded application. At the beginning of a checkpoint, iDNA records the architectural state consisting of the values in the registers and the program counter. And then during the program's execution, iDNA dynamically instruments the load instructions and records their values. The log size generated is reduced using a compression mechanism [3].

Recording the values of load instructions executed by a program automatically takes care of all forms of non-determinism, including system interactions (system calls, interrupts, DMAs) and multi-threaded interactions (even when multiple threads are executing on multiple processors). For example, if a system call or an interrupt modifies a memory location, the program needs to load the value from the memory location before it can use the value. Therefore, recording the values of the load instructions is sufficient to capture the system interactions. Even DMAs that concurrently modify the program's memory state can be taken care of by logging the load values. Also, in the case of multi-threaded programs, multiple threads can be concurrently modifying a shared memory location, but as long as we record the load values for a particular thread, we can replay that thread.

Note, iDNA does not log every load value. It records only the load that accesses a memory location for the first time. In addition, if a memory value is modified by the external system (DMA, system call, another thread, etc.) outside of the thread, then the value of the subsequent load to that location is logged. iDNA also correctly deals with the dynamically loaded libraries and self-modifying code. A detailed explanation on how all this is done efficiently can be found in [3].

3.2 Sequencers for Multi-Threaded Programs

In the case of multi-threaded programs, a replay log is recorded for each thread individually. As we described above, the replay log for a thread contains the initial architectural state of the thread and all the load values that are necessary to replay that thread correctly. Even if other threads are concurrently modifying the shared memory locations, it does not affect how a thread is replayed. Because, iDNA logs the values of all the required load instructions in the replay log. Thus, using the replay log of a thread, we can replay the thread exactly how it was executed during the original execution.

However, to aid debugging and to enable multi-threaded program analysis, we want the ability to deterministically replay the thread interactions observed during the original execution. iDNA provides this functionality by recording what are called *Sequencers*. A sequencer log consists of a global time-stamp value that is maintained by iDNA (one global time-stamp counter maintained across all the threads). The global time-stamp is incremented whenever a sequencer is logged in the replay log of any thread.

A sequencer is recorded when a synchronization instruction or a system call is executed. iDNA dynamically instruments the instructions with the lock prefix to recognize the synchronization operations. Whenever a synchronization operation is executed by a thread, a sequencer is logged. Since each sequencer consists of a time-stamp that is incremented monotonically, there exists a total order between all the sequencers recorded across all the threads. Figure 1 shows an example for how sequencers are recorded in the replay log of each thread. The sequencers are labeled as S1, S2, etc. For the example, assume that $Si > Sj$ if $i > j$. With this log, we can determine that all the memory operations that were executed before the sequencer S1 in the thread T2, should have been executed before all the memory operations that were executed after the sequencer S3 in the thread T1 (because time-stamp for S3 is greater than S1).

3.3 iDNA Replayer

To replay a thread using a replay log, first the architectural state of a thread comprising of the registers and the program counter are initialized with the information read from the log. iDNA records both the code and the data that is read by a thread. So during replay, the execution starts from the instruction pointed to by the program counter. The load instructions are then dynamically instrumented so that the iDNA replayer can make sure that replayed loads return the correct values as recorded in the replay log.

In the case of multi-threaded programs, one sequencing region is replayed at a time. A sequencing region consists of the sequence of instructions executed between two consecutive sequencers logged in an iDNA log for a thread. For the example in Figure 1, the instructions executed between $S3-S5$ constitute a sequencing region. One sequencing region is replayed at a time and it is chosen from one of the thread as follows. The sequencing region that has the smallest starting sequencers among all the sequencing regions that are yet to be replayed is chosen for replay. For example, in the Figure 1, the sequencing region $S1 - S4$ is replayed before $S2 - S6$. After replaying $S2 - S6$, the region $S3 - S5$ is replayed and so on.

3.4 Finding Happens-Before Data Races

Using iDNA, we replay a multi-threaded program's execution using the above sequencers. During replay, we modified iDNA to analyze the program's execution to find data races between sequence regions.

To find the data races, we use the sequencers recorded in the iDNA traces. Using the sequencers, we can determine the overlapping sequencing regions across different threads in a multi-threaded program execution. For example, in the Figure 1, the instructions executed between $S3 - S5$ constitute a sequencing region. It overlaps with the sequencing regions $S1 - S4$ and $S4 - S7$ in the thread T2, and also with the sequencing region $S2 - S6$ in the thread T3. In other words, there is no happens-before relationship

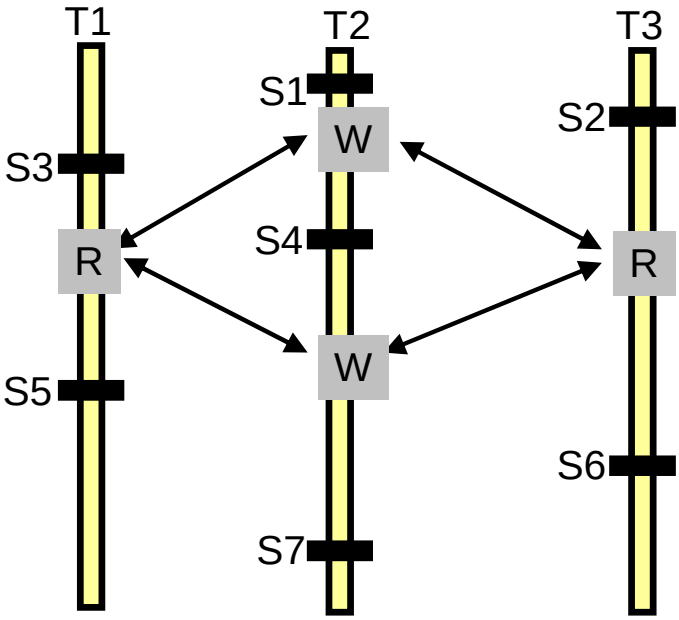

Figure 1. Happens-before based race detection during replay using sequencers in the replay log.

between the memory operations executed in the overlapping sequencing regions.

We then detect a data race using the following happens-before algorithm. If we find two memory operations in two overlapping sequencing regions, and at-least one of them is a write, then we consider that the two memory operations to be involved in a data race. There is a data race between those two memory operations, because there is no sequencer separating the two in time to specify an order between them. If there is no sequencer between two memory operations, then it implies that there was no synchronization operation that was executed during the program's execution to guard the shared memory accesses. Therefore, there is a data race between the two memory operations.

4. Classifying Data Races by Replaying Both Orderings

In the previous section, we described how we find a set of data races in a given program's execution by looking for memory operations that are not ordered by a happens-before relation. In this section, we present a replay-based dynamic analysis algorithm that automatically classifies data races as either potentially benign and potentially harmful.

4.1 Overview

Consider a data race between two memory operations executed in a particular execution of a multi-threaded program. The two memory operations involved in the conflict would have been executed in a particular order during the original execution recorded by iDNA.

During replay, there are two possible orderings for the two memory operations involved in the data race. Our analysis, experiments by replaying the two different orders for those memory operations and examines the outcome of the two executions. If the outcomes are the same, then we classify that instance of the data race as potentially benign. Otherwise, the data race is classified as potentially harmful, which will then be examined by the developer.

Figure 2 shows a piece of code to illustrate how the proposed analysis works. This is a sanitized example of one of the harmful

```
Thread 1 and 2
executes:
{
    foo->refCnt--;

    If(foo->refCnt == 0)
        free(foo);
}
```

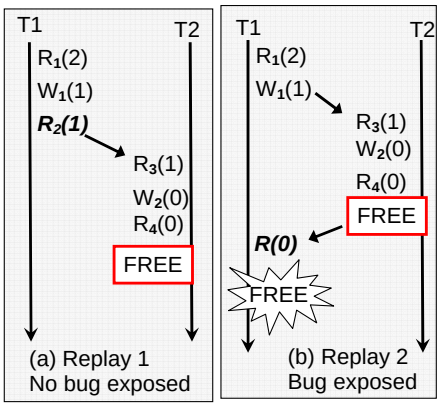

Figure 2. Race Detection Example

data races found during our analysis on production code. The example code essentially decrements a reference counter value. Then, it reads the reference counter value. If the value is zero it frees the memory pointed to by the variable "foo". Assume that there are two threads executing the same piece of code in parallel and that the programmer, by mistake, did not use any synchronization operations to guarantee the correct parallel execution.

The figure also shows two possible orderings for the memory operations when this piece of code is concurrently executed by two threads. The values of the memory operations are shown inside the parenthesis. Figure 2(a) shows the order observed during recording. Fortunately, for this ordering, the atomicity of the operations was not violated and hence the program executes correctly. However, during our dynamic analysis, we will detect data races between the read and the write operations executed in the two threads. For example, there is a data race between the read R2 in thread T1, and the write W2 in thread T2. During dynamic analysis, we can replay for the two possible orders between these two memory operations. One replay will be the same as the one observed during the original execution that was recorded as shown in Figure 2(a). Another possible order is shown in Figure 2(b). In the latter order, the R2 is replayed after W2. During that replay, we will catch a null pointer violation, when the replay tries to free the location "foo" in the thread T1. Thus, we determine that the data race is potentially harmful.

In contrast, when we examine the two orderings, if they evaluate to the same result, then we classify that instance of the data race as potentially benign. However, even if one of the instance of the data race was found to be potentially harmful, then we classify the data race to be potentially harmful. We next describe how we perform our replay analysis to determine if the two orderings arrive at the same result or not.

4.2 Mechanism for Alternative Replay

The above discussion assumes that it is possible to replay the two possible memory orders. We had to add the following support to provide this functionality on top of iDNA.

Our algorithm analyzes each data race in isolation. For a given data race, the goal is to replay the two possible memory orders between the two memory operations involved in the data race. The two memory operations involved in the data race are part of two sequencing regions in two different threads (a sequencing region constitutes the instructions executed between two sequencers in a thread, which we described in Section 3).

By replaying the program's execution, we can determine all the instructions executed in each of the two sequencing regions that contain the data race. Using this information, we know which two dynamic instructions are the data race being considered, and we can examine both orders of those two instructions during replay. We replay both threads for the region up until we get to the data race instruction in each thread. We then can replay the two orders to examine the differences exhibited by the data race. The first order we call the *original order*, since it matches the values seen during the original logged execution, and the second order we call the *alternative order*.

In order to execute the instructions in the two sequencing regions for the two orders, we added to iDNA the ability to create a virtual processor. The virtual processor allows us to start with a set of sequences across the threads and execute multiple different realities starting at that set of sequences. A virtual processor is created to execute the original and alternative replay orders. The virtual processor is initialized with the live-in memory values and the register states of the two threads. We orchestrate the execution of the two threads in the virtual processor to obey the ordering for the instructions involved in the data race. Whenever a memory location is read for the first time in the virtual processor, the virtual processor copies the value from the live-in memory. Then from that point on, the reads and writes to that memory location will be to the local copy in the virtual processor.

4.2.1 Alternative Replay Failure

While executing the alternative ordering, the replayer may come across a memory reference to an address not seen when the original log was taken, or it may come across a control flow change. The address may not have been logged or it may have been changed during replay, so we do not know what the value is. For the control flow change, it may jump to a piece of code that was not recorded as part of the logging or to an illegal address. In our current implementation, we classify all of these as *replay failures*. They are an indication that execution has changed enough from the alternative order that the data race is potentially harmful. Even so, we are looking at trying to log enough information to allow replay to continue in the face of both of these for the final version of our tool.

4.3 Classifying Data Races

After we have replayed the original and alternative orderings in the virtual processors, we compare the register and memory live-outs at the end of the sequencing regions to classify the data race as potentially benign or potentially harmful.

A data race between two memory operations may occur many times during our analysis, and we examine each of those as a separate data race instance. Our current approach flags a data race instance as potentially benign only if the two replays result in exactly the same application state (both memory live-outs and register state) at the end of the replay. Otherwise, the data race instance is considered to be potentially harmful. The potentially harmful consist of the data races where the alternative replay resulted in different state, and also those that had a replay failure as described above.

After all of the instances for a data race have been examined, we classify the data race as potentially benign only if all of its instances are classified as potentially benign. Otherwise the data race is classified as potentially harmful.

The data races classified as potentially benign are guaranteed to be benign for the test scenarios we examined, but they are not guaranteed to be benign for all possible scenarios. Another instance of the data race not captured in our replay logs between the same two memory operations may prove to be harmful. To add more confidence to our classification, several instances of the same data race should try to be found in the same execution or across the different test scenarios. If the replay analysis determines the data race to be potentially benign in all those instances, then we will have greater confidence that the data race is probably benign. The greater the number of instances studied, the greater is the confidence that a data race is benign.

For those data races that are classified as potentially harmful the two replays will enable the developer to have a better understanding of the effects of the data race that is reported. The two replays will show the differences in the outcomes of the program for the two different memory orders between the memory operations involved in the data race.

4.4 Advantages

There are some advantages with our replay based analysis:

- Our analysis is at the instruction level and is not dependent on the specific synchronization methods. As a result, it is applicable to programs written in any language as it is agnostic to the synchronization methods used in the language. Our instruction based happens-before analysis does incur a heavy performance overhead. However, this is not a serious concern for our approach because we perform our analysis off-line during replay, where we can take more time to do the dynamic analysis.
- Unlike traditional approaches where it is hard to determine the possible effects of a data race, we will have two possible executions for the data race and produce the output for those executions. The ability to replay and see the differences in output between the two executions is of great value for the developer to understand the potential data race.

5. Results

In this section, we discuss our experiences in using our tool to classify the data races.

5.1 Methodology

We collected replay logs for 18 different executions of various services in Windows Vista and the Internet Explorer. Among the 18 executions that we studied, the happens-before based algorithm that we described in Section 3 returned 16,642 instances of data race conflicts. Out of these 16,642 instances there were only 68 unique data races. The reason is that a data race (between the same two memory instructions in different threads) occurred more than once in the same execution or in different scenarios. For this study, we went through the painstaking effort to manually examine every single data race to determine if it was actually benign or harmful. All of the data races that were identified as truly harmful have been fixed in the production code.

The average log size for the replay logs collected using iDNA was about 0.8 bit per instruction. The total storage space required for the logs was 3.1 GB, which captured 33 billion instructions executed across all the different executions that we studied (about 96 MB to record a billion instructions). By compressing the log sizes using the Windows `zip` utility, we reduced the log sizes to about 0.3 bit per instruction.

To get an estimate for the time overhead for recording, replaying and analyzing programs we studied an execution of Internet Explorer, where we accessed a website and browsed through a few pages. This study was carried out on a Pentium 4 Xeon 2.2GHz processor with 1GB RAM. The runtime performance overhead to collect the replay logs using iDNA [3] was about 6x when compared to the native execution. The iDNA replayer can replay the recorded execution with a performance overhead of 10x on average (relative to the native execution). The execution had spawned

	Potentially Benign		Potentially Harmful		
	Real Benign	Real Harmful	Real Benign	Real Harmful	Total
No State Change	32	0	-	-	32
State Change	-	-	15	2	17
Replay Failure	-	-	14	5	19
Total	32	0	29	7	68

Table 1. Data Race Classification

27 threads. When we ran our happens-before based race detection analysis, we found 2,196 instances of various data races. The overhead of executing the off-line happens-before race detection analysis was about 45x. The overhead of executing the replay analysis that we described in Section 4 to classify the data races was about 280x when compared to the native execution.

5.2 Data Race Classification Results

5.2.1 Outcomes of Replay Analysis

We performed the replay-based data race classification analysis that we described in Section 4 over all the instances of the data races that were found using the happens-before algorithm. There are three possible outcomes when we perform the replay based analysis for an instance of a data race. The two replays for an instance may produce the same live-out. We call this outcome *No-State-Change*, because the memory order does not affect the state of the program's execution. Another possible outcome is that the two replays might produce different live-outs. We call this outcome *State-Change*. Finally, for some instances of data races we may encounter a replay failure while replaying for the alternative order for the reasons that we described in Section 4. We call this outcome *Replay-Failure*. Note, that a replay failure is a good indicator that the data race is likely to cause a change in the program's state (in other words, the outcome is similar to State-Change).

There can be many instances for a given unique (static) data race. The final classification for a data race classifies the data race as No-State-Change only if all of its instances are No-State-Change. If for any instance of the data race, the outcome was a State-Change, then we place the data race in the State-Change group. All of the remaining unique data races are classified as Replay-Failure. These are the data races for which none of the instances were classified as State-Change and at least one of the instances was classified as Replay-Failure.

5.2.2 Data Race Classification

Table 1 presents the classification for all the unique static data races (a data race between the same two static instructions) that we studied. The rows in the table correspond to one of the three outcomes of the automatic replay analysis that we just described.

Based on the outcomes of the replay analysis for all the instances of a data race, our replay checker classifies the data race as either *Potentially-Benign* or *Potentially-Harmful*. These two classifications are shown in the table as the two aggregate columns. All data races classified as No-State-Change are potentially benign, and all data races classified as State-Change or Replay-Failure are classified as potentially harmful.

Table 1 splits the potentially benign and harmful columns further into two groups: *Real-Benign* and *Real-Harmful*. The sub-columns correspond to the manual classification. In addition to the automatic classification, we also manually triaged each data race to determine if they were really benign or harmful. This was done to determine the accuracy of the automatic classification.

5.2.3 Potentially Benign Data Races

Table 1 shows that out of the total 68 data races that we studied, 32 data races fell into the No-State-Change group. Since none of the instances of these data races can cause a state change or a replay failure, our automatic analysis classified these 32 data races as potentially benign. We manually verified each of these data races and found that they were all indeed benign. None of them were found to be harmful.

5.2.4 Potentially Harmful Data Races

The data races accounted for in the second and the third rows in the Table 1 were classified as potentially harmful. The reason behind this classification is that, in at-least one instance of a data race, if the outcome of the replay analysis was either a state change or a replay failure then it has the potential to be harmful. Based on this classification the automatic replay analysis classified 36 data races (17+19) to be potentially harmful.

Seven among the 36 data races were found to be harmful through manual inspection, as listed in the sub-column named Real-Harmful under the Potentially-Harmful column. The automatic analysis correctly classified all the real harmful data races that it analyzed as potentially harmful. Two of these harmful data races are similar to the reference counting example that we discussed in Section 4.

However, as we can see from the table not all of the potentially harmful races were found to be harmful in our manual classification. The sub-column named Real-Benign under the Potentially-Harmful column shows that 29 data races that were classified as potentially harmful are actually benign. The following are the two main reasons for the misclassification.

Misclassification Due to Approximate Computation: By manually inspecting these 29 data races, we found that 23 of them actually affect the execution of the program. As a result, our replay analysis will find a state change or a replay failure for most of the instances of these data races. Therefore, they were classified as potentially harmful. We took these potentially harmful data races to the developers. They described that these data races were left in the production code, because they chose to tolerate the effects of the data race rather than synchronize the code and lose performance. A good example where this kind of optimization is possible is the code region that was used to update a data structure maintaining statistics. In that case, the programmer consciously chose to gather approximate statistics and avoid the performance overhead required to accurately gather them. Another example is where the variable's value is used to make decisions that can affect only the performance and not correctness (e.g., time-stamp value used for making decisions on what to replace from a software cache). To optimize the synchronization overhead, programmers may choose to not synchronize operations on values such as time-stamps and statistics wherever appropriate. Since these data races were intended by the programmers, the are classified as Real-Benign, even though they can change the program's execution.

Misclassification Due to Replayer Limitation: We now focus on the remaining 6 Real-Benign data races of the 29 data races

that were incorrectly classified as Potentially-Harmful. When we manually triaged the six data races, we found them to be benign. Unlike the other 23 data races that we discussed earlier, these six data races did not affect the output or the state of the program. The reason why these six data races still got classified as Potentially-Harmful is that for at-least one of their instances, the outcome of the replay analysis was Replay-Failure. The replay failure was to due to the reasons that we described in Section 4.2.1. When we manually analyzed these 6 replay failures, we actually found that the execution of the program wouldn't have been affected had the replays proceeded without failing. By adding additional support in iDNA to execute down unseen control paths, we should be able to correctly classify these six data races as no-state change and thereby classify them as potentially benign.

In conclusion, our approach classified 47% of the data races as potentially benign and they were all benign (none of them were harmful). Out of the other 53% of the data races that were classified as potentially harmful, only 20% of the 53% were found to be harmful.

5.3 Results for Each Dynamic Data Race Instance

Let us now discuss the results for the each of the instances that we analyzed for every static data race. We will also discuss the type of outcome that we obtained from the replay analysis for each instance.

Figure 3 shows the number of instances that we analyzed for each of the 32 data races that were of the type No-State-Change, which we classified as Potentially-Benign. The number of instances for each unique data race varied from about 50 instances to just one instance. The greater the number of instances that we analyze and classify as No-State-Change, the greater the confidence we have in classifying them as Potentially-Benign.

Figure 4 shows the number of instances that we analyzed for each harmful data race. As we can see, for some of the harmful data races we analyzed several thousand instances. However, only one in ten of those instances caused a replay failure or a state change. This shows that it is important to see those data races multiple times in order to catch them as Potentially-Harmful.

Figure 5 shows the number of instances that we studied for the data races that we considered to be Potentially-Harmful, but when we analyzed them manually we found them to be Real-Benign. The main cause for this misclassification are the data races due to approximate computation, which we described in Section 5.2.4.

5.4 Reasons for Benign Data Races

In this section, we describe the categories of benign data races that we were able to automatically classify as potentially benign.

1. **User Constructed Synchronization:** Programmers may construct their own synchronization primitives without using fences or the atomic operations provided in the instruction set architecture. For example, a garbage collector can maintain the reference counts for concurrent objects without using locks [9]. It is difficult to automatically infer the user constructed synchronization operations and so iDNA does not log a sequencer for a user constructed synchronization operation during the logging run. Because of this reason, the happens-before algorithm, will incorrectly classify a race between two user constructed synchronization operations, which is essentially correct synchronization, as a data race.
2. **Double Checks:** Double checks are used to optimize the synchronization overhead. A typical example for a double check is:

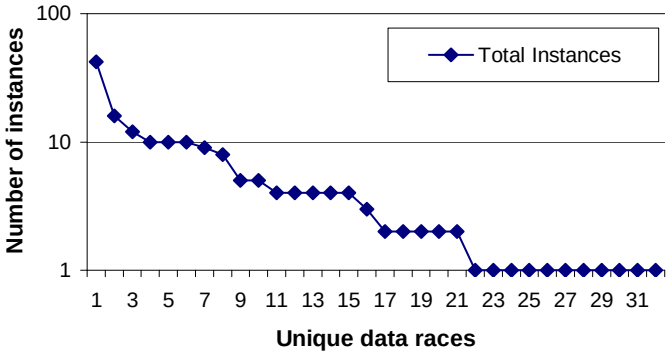

Figure 3. Statistics for the unique data races classified as Potentially-Benign. Every instance of these data races resulted in No-State-Change and were actually Real-Benign. Total number of instances for each such data race are shown.

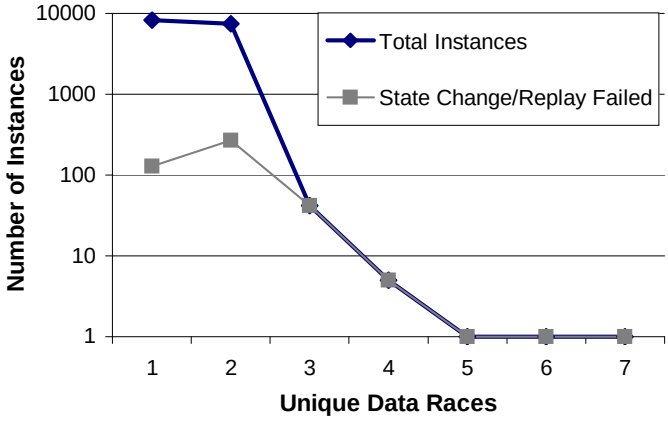

Figure 4. Statistics for the unique data races that were classified as Potentially-Harmful and they were found to be Real-Harmful. Results are shown for total number of instances, and also for the number of instances that resulted in a State-Change or Replay-Failure.

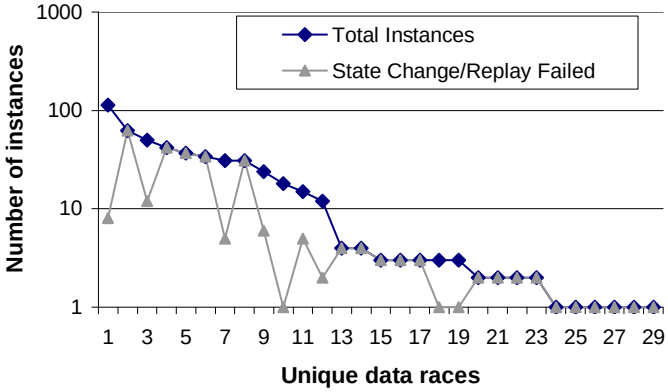

Figure 5. Statistics for the unique data races that were classified as Potentially-Harmful, but they were actually Real-Benign. Results are shown for total number of instances, and also for the number of instances that resulted in a State-Change or Replay-Failure.

	# Races
User Constructed Synchronization	8
Double Checks	3
Both Values Valid	5
Redundant Writes	13
Disjoint bit manipulation	9
Approximate Computation	23

Table 2. Benign Data Races.

```
if(a) {    lock (..) { if(a) ... } }
```

The read in the first check is not synchronized and so there can be a data race involving the read, but the data race is benign.

3. **Both Values are Valid:** Let us consider a data race between a read and a write operation. We found many instances where it is correct for the read operation to return either the old or the updated value (old value is the value in memory before the write and updated value is the value in memory after the write).

   For example, when a buffer is shared between the producer and the consumer it can be correctly synchronized without using synchronization primitives. The producer writes to a buffer and increments the number of writes. The consumer reads the number of writes, and if it is greater than the number of entries it has consumed so far (referred to by a local variable), then it consumes an entry from the buffer. After consuming a value it updates its local variable representing the number of entries consumed. Without explicit synchronization, it is possible that the consumer might read a stale value for the buffer size. But that is fine, since it will just force the consumer to wait longer.

   In another example, a shared variable was checked to decide which of the two versions of a function need to be used for doing a particular computation. Both the functions do exactly the same computation, but with different performance characteristics. The shared variable is written by another thread to specify the version of the function to be used. However, the read and the write need not be synchronized, because both versions will produce correct results, though one version might perform slower than the other.

   Similar to this, we found the case where it just mattered if the memory value zero or non-zero. The code was valid for multiple writers setting the memory value to non-zero without any synchronization, and it did not matter if the non-zero value written was the same.

4. **Redundant Writes:** If a write operation writes the same old value that already resides in the memory location then the data race between the write and a read operation in another thread will be benign (thus it can be considered as a special case of the previous category in the sense that both the old and the updated values are correct values to return for a read operation). In one of the programs we studied, we found that a thread was writing its process identifier returned by a system call to a shared variable read by another thread. The writes were redundant and did not affect the correctness of the program execution.

5. **Disjoint Bit Manipulation:** There can be data races between two memory operations where the programmer knows for sure that the two operations use or modify different bits in a shared variable. Programmers tend to use multiple bits in the same variable in order to optimize for performance.

Table 2 shows the number of data races that we studied for each of the above categories of benign data races. It also shows that there were 23 data races that were due to approximate computation, which were mis-classified by the replay analysis. As we mentioned in Section 5.2.4, there were six other benign data races that were misclassified as Potentially-Benign. These six were due to those benign data races that can affect the control flow of the program's execution. The rest of the benign data races were correctly classified as Potentially-Benign and the reasons for why they were benign are shown in Table 2.

6. Conclusion

In this paper, we focused on automatically finding the potentially harmful data races. The happens-before algorithm that we used does not report false positives, but it still yields a large number of true data races, out of which 90% are benign. To reduce the triage effort, we needed to automatically identify and filter the data races that are potentially benign.

We built our dynamic analysis mechanism on top of iDNA, which provided us the ability to record and replay a program's execution. To automatically find out if a data race is potentially benign or not, the replay based checker replays the execution twice, one for each possible order between the conflicting memory operations. If the two replays for the two orders produce the same result, then the checker classifies the data race as potentially benign.

In addition to reporting harmful data races, the analysis also produces very useful information to assist a programmer in debugging the data race. For every data race, the checker dumps out the replay log along with the memory orders corresponding to the data race. Using that information, a programmer can replay the program in two different ways and understand the effects of different memory orders that are possible due to the data race. This information can be a significant aid to the developer.

We discussed our experiences in using our dynamic race classification approach on an extensively stress-tested build of Microsoft's Windows Vista and Internet Explorer. Our proposed technique was able to automatically filter out over half of the real benign data races, by classifying them as potentially benign, which can be ignored by the developers. In addition, all of the real harmful data races were correctly classified as potentially harmful. The harmful data races that we found were reported to the developers and all of them have been fixed in the production code.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on this paper. This work was funded by grants from Intel and Microsoft.

References

[1] 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.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, 2005.

[3] S. Bhansali, W. Chen, S. de Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*, June 2006.

[4] 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.

[5] 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, New York, NY, USA, 2002. ACM Press.

[6] 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.

[7] J. D. Choi and S. L. Min. Race frontier: reproducing data races in parallel-program debugging. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–154, 1991.

[8] 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.

[9] D. L. Detlefs, P. A. Martin, M. Moir, and Jr. G. L. Steele. Lock-free reference counting. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 190–199, 2001.

[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] T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI*, 2005.

[13] 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.

[14] 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.

[15] 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.

[16] M. Hicks, J. S. Foster, and P. Pratikakis. Inferring locking for atomic sections. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

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

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, 2006.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[28] 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.

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

[30] M. Ronsse and K. de Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 5 1999.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[38] 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.