CoRD: A Collaborative Framework for Distributed Data Race Detection

Baris Kasikci, Cristian Zamfir, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Modern concurrent software is riddled with data races and these races constitute the source of many problems. Data races are hard to detect accurately before software is shipped and, once they cause failures in production, developers find it challenging to reproduce and debug them.

Ideally, all data races should be known before software ships. Static data race detectors are fast, have few false negatives, but unfortunately have many false positives. Conversely, dynamic data race detectors do not have false positives, but have many false negatives and incur high runtime overhead. There is no silver bullet and, as a result, modern software still ships with numerous data races.

We present CoRD, a collaborative distributed testing framework that aims to combine the best of the two approaches: CoRD first statically detects races and then dynamically validates them via crowdsourced executions of the program. Our initial results show that CoRD is more effective than static or dynamic detectors alone, and it introduces negligible runtime overhead.

1 Introduction

Data races are a common root cause for many concurrency-related problems [20]. They caused the loss of human lives [18] as well as material losses [27]. Moreover, practitioners report that it typically takes weeks, sometimes even months, to diagnose and fix a data race [9], which means that data races also waste significant development resources.

Even though they have potentially catastrophic consequences, data races are prevalent in modern software. Running a state-of-the-art data race detector while loading a web page in Firefox causes the detector to report more than $1,000$ unique data races [14]. The reason why software riddled with races still works is that, data races do not always lead to an observable harmful effect in real applications (5%-24% according to [13, 14, 22, 28]). This is because most data race bugs occur under thread interleavings that are rarely encountered during the common execution case.

Ideally, programs should have no data races at all. For instance, the new C and C++ standards allow the compiler to perform certain optimizations that may transform seemingly benign data races into harmful ones [2]. The question is then how to ensure that software does not ship with any data races? This can be achieved by using data race detection before software is shipped.

Data race detection can be broadly classified as either static or dynamic. Static data race detectors [5, 28] analyze the program source code without executing it. They scale to large code bases, providing results in a short amount of time. For instance, RELAY can analyze and report races in the Linux kernel (4 MLOC) in around 5 hours. Static race detectors typically have fewer false negatives (i.e., do not miss real races) than dynamic race detectors [23]. However, static race detectors tend to have many false positives (i.e., produce reports that do not actually correspond to real races). For example, 84% of the races reported by RELAY [28] are not true races.

Dynamic data race detectors typically do not have false positives, but only detect data races in executions they can directly observe, therefore they have many false negatives. They also have high runtime overhead (e.g., $200\times$ in the case of Intel ThreadChecker [11] and up to $8.5\times$ in the case of FastTrack [6]), because they typically need to monitor all memory accesses and synchronization primitives. Detectors that employ sampling [13] decrease runtime overhead at the expense of introducing both false positives and false negatives. High runtime overhead indirectly decreases the coverage of dynamic detectors: they cannot be enabled in production, so they are only used by developers to detect races in executions of the program's test suite.

Therefore, the use of data race detection is limited on the one hand, by the abundance of false positives of static race detectors and, on the other hand, by the large number of false negatives and prohibitive overheads of dynamic race detectors. The problem stems from how the two approaches analyze program paths: Static detectors cannot obtain the full path context needed to determine the validity of the race, while dynamic detectors cannot perform the full-program analysis needed to find all races. This is why existing race detectors are rarely used [21].

We introduce CoRD, a collaborative framework for data race detection that combines the benefits of static

and dynamic detectors: it has few false positives, few false negatives, and incurs negligible runtime overhead. CoRD detects data races using a static analysis pass with few false negatives, and then it crowdsources the dynamic validation of these data races in a full-path context on many machines, using a collaborative distributed framework. The validation step confirms real data races (true positives) and identifies likely false positives. CoRD is lightweight, and thus suitable for use in production software. Additionally, CoRD explicitly increases the probability of manifestation of race-related failures for testing purposes, thus helping developers to promptly classify the consequences of real data races [14] and hopefully getting them fixed before they affect a wider user population.

In preliminary experiments, CoRD effectively detects 3 races that lead to deadlocks in SQLite and 4 races that lead to crashes in Pbzip2, while incurring a maximum runtime overhead of 1.6%.

2 The Case for Collaborative Race Detection

The collaborative race detection framework is a concrete instance of the SoftBorg information recycling vision [3], which argues that nodes running copies of the same program P should collaborate in identifying and fixing incorrect code in P, as well as construct proofs of the correct code.

Rather than trying to automate testing using a single, in-house testing infrastructure, SoftBorg advocates distributing the testing tasks among its participants in a way that is efficient and minimally intrusive for the participants. If the community of cooperative nodes is large, testing should be comprehensive and should amortize the overhead across the participants. The testing results should be more "meaningful" than in-house testing, because they result from actual program executions of the participants.

SoftBorg leverages the observation that software users have collectively, overwhelmingly more hardware than any single software company. For instance, Google's Chrome Web browser runs on more than 300 million computers [10], which exceeds by more than two orders of magnitude the most optimistic estimates of how many servers are housed in Google's data centers [12]. Therefore, leveraging end-users to perform testing offers substantial advantages.

In this paper, we present an initial attempt at materializing this vision for the case of distributed collaborative data race detection. We describe an early-stage design and prototype, along with promising preliminary results. We also discuss our future research directions.

3 CoRD Design

CoRD's high-level architecture is shown Figure 1: The hive is a central program that performs data race detection. Pods are daemons running at the end-users' site, and they monitor the executions of program P and relay by-products of the execution to the hive.

We envision CoRD being used in the following way: First, developers set up a hive service for their software. The hive statically detects races in some program P and produces an instrumented version of P. The users download the instrumented version of P together with the pod. Then, the hive instructs the various running instances of P (by communicating with their pods) to explore both possible interleavings that a pair of racing memory accesses can exhibit. We call these interleavings the primary and the alternate. If the hive and the pod manage to orchestrate P to follow both the primary and the alternate interleaving, we say that the race is validated, because there exists conclusive proof that it is a real race.

The pods automatically connect to the hive and report the explored interleavings. The hive validates the races based on these reports. Validation results are then inspected by developers, who eventually fix the validated races and update the software. A new (instrumented) version of the software is created, and the process repeats.

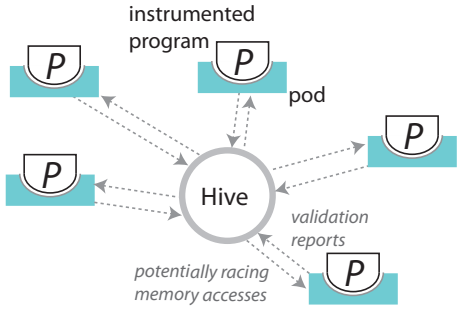

Figure 1. CoRD general architecture

3.1 The Hive

The hive is set up and administered by the developers of an application. It can be deployed in a centralized server or in the cloud, depending on the required computational power. The hive generates instrumented binary executables, and makes them available to the users for download (Figure 2): (step 1) the hive takes the program source code as input and compiles it to LLVM bitcode; (step 2) a static data race detector, operating on LLVM bitcode, identifies pairs of potentially racing memory access instructions; (step 3) the hive instruments the LLVM bitcode with calls before and after the potentially racing instructions. This instrumentation is used to orchestrate the thread schedule at the end-user site.

The instrumentation is done for all the potentially racing accesses, however, the hive selectively activates it on-demand when P runs, in different ways for different end-user machines. Essentially, this activation mechanism aims to validate as many races as possible by uniformly distributing validation tasks across the participants to CoRD. Finally, (step 4) the hive compiles the

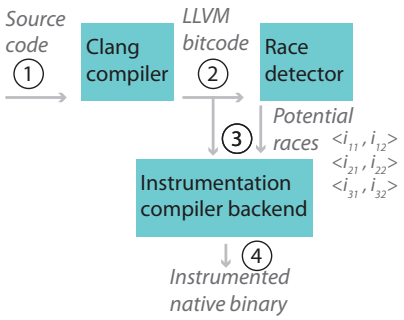

Figure 2. CoRD hive architecture

instrumented LLVM binary to the specific architecture of the end-user machine (e.g., x86, ARM).

CoRD's static race detector uses an interprocedural, flow-sensitive analysis that implements a static variant of the popular lockset-based dynamic race detection algorithm [25]. This algorithm tries to infer whether, for any pair of accesses to a shared variable, of which at least one is a write, the accesses can occur without the threads holding a common lock. If this is the case, the algorithm flags a race. Our algorithm is based on the algorithm used in RacerX [5], modified to have fewer false negatives.

CoRD's instrumentation compiler backend generates lightweight instrumentation calls to the pod that allow the pod to orchestrate the thread schedule on the user machine. More details follow in §3.4.

3.2 The Pod

Once a hive–pod connection is established, the hive sends the pod a target race and the desired order of racing accesses (i.e., the primary or alternate). The pod monitors the instrumented binary and enforces the program to follow the order of the racing accesses specified by the hive. To achieve this at runtime, the pod intercepts the instrumentation calls that were generated by the hive and orchestrates program schedule (§3.4).

The pod then relays the information gathered during schedule orchestration back to the hive for it to validate races. The pod can send three types of reports: (1) schedule enforced, (2) schedule not enforced, or (3) specification violation (crash, deadlock, or assert). These reports are used in race validation, which we detail below.

3.3 Race Validation

CoRD validates data races based on the reports received from the pods. In this section, we detail how this validation is performed.

3.3.1 Validating True Positives

CoRD confirms true positives (real data races) as follows: if both the primary and the alternate schedule of a potential data race can be enforced in the same execution context, then it means that there is no mutual exclusion between the two accesses. Thus this is a data race. However, this data race may not necessarily lead to a specification violation. The ability to confirm potential races before reporting them is a key component of CoRD's improved detection accuracy.

3.3.2 Gathering Evidence for False Positives

CoRD cannot tell with certainty if a data race is a false positive or not, but it uses two heuristics to report likely false positives. The first heuristic monitors whether only one ordering of the potentially racing accesses can be enforced. This is usually the case if the racing accesses are properly synchronized. The static race detector flags such accesses as races because it may not be aware of some synchronization primitives. The second heuristic monitors the purported racing memory accesses and, if they do not access the same memory, the data race is deemed a likely false positive.

CoRD leverages its large scale to increase the confidence that a report is a potential false positive. The more instances of a likely false positive race report are reported to the hive as a result of different executions, the higher the probability that the race report is actually a false positive.

3.3.3 Reporting Specification Violations

CoRD also increases the likelihood of triggering race-induced failures that violate the program specification (e.g., crash, deadlock, or assert).

In the case of a crash, the pod catches the SIGSEGV signal and submits the crash report to the hive. CoRD takes a similar approach in the case of a hang or when the program receives an unhandled SIGINT (e.g., the user pressed Ctrl-C). In this case, CoRD prompts the user with a dialog asking whether the program has failed to meet expectations. If yes, the pod informs the hive that the enforced schedule leads to a specification violation.

3.4 Schedule Orchestration

The pod attempts to enforce the instrumented binary running at the end-user site to follow either the primary or the alternate schedule, as instructed by the hive. If the program follows the desired schedule by itself, without the pod having to steer it, CoRD's thread schedule orchestration is idle and has virtually no overhead. However, if the program is about to exhibit the reverse schedule (primary instead of alternate, or vice-versa), the orchestration kicks in to enforce the required order.

The case in which the orchestration takes over is shown in Figure 3. Assume a program with two threads, T_1 and T_2, and two racing accesses, r_{11} and r_{12}. Let's further assume that the hive instructs the program to follow the primary schedule of racing accesses, which in

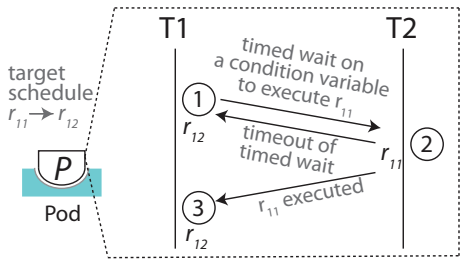

Figure 3. Schedule orchestration

this case corresponds to the order $r_{11} \rightarrow r_{12}$ of racing accesses, meaning that r_{11} executes before r_{12}. Assume that during execution, T_1 is about to execute r_{12} before r_{11}. Since this is not the order desired by the hive, the orchestration preempts T_1 for a configurable period of time (using timed wait on a condition variable), allowing T_2 to get scheduled and execute r_{11} (step 1).

If T_2 executes r_{11} (step 2), the instrumented code signals (using a condition variable signal statement) T_1 to continue and execute r_{12} (step 3). Once r_{12} is executed, the pod notifies the hive that the required schedule was enforced.

On the other hand, if the timed wait timeouts as shown in Figure 3, and T_2 does not execute r_{11}, T_1 gets scheduled back and r_{12} gets executed. This could mean two things: (a) the timeout was too short for T_2 to have a chance to execute r_{11}, or (b) the interleaving in question cannot be enforced under any circumstances (perhaps because there is some ordering constraint between the accesses such as an ad-hoc synchronization [30], that the static race detector at the hive did not recognize). Either way, the pod notifies the hive that the required order cannot be enforced.

When CoRD enforces a particular schedule, the upper limit on the overhead is the timeout duration. Similarly, if the schedule cannot be enforced, the overhead stems from the unnecessary wait.

This constant upper bound on the timeout is key in achieving low overhead. To preserve the low latency of interactive applications, CoRD uses short timeouts of less than 200 ms; for server applications, higher timeouts can be configured by developers. It is also possible to configure the value of the timeout depending on how often the potentially racing accesses execute.

3.5 Incentives for Using CoRD

Today, many users experience the same concurrency problems, and these problems go unnoticed. End users may experience crashes, hangs, data loss, etc. while validating the races, in cases in which they would not have encountered such behavior. However, if one user encountered this problem, then many other users will not, therefore we sacrifice one user for the good of many.

To motivate users to participate, we envision using a reward system for users who find true positives. Such users can be allowed free upgrades and free product trials if the software under test comes from a vendor, or they can be awarded "badges of honor" in the case of open-source software.

This system could work well for beta testers (Windows 7 had more than 8 million beta testers [1]) and also for regular users. We believe that CoRD's low runtime overhead is key in increasing adoption by regular users.

4 Preliminary Experimental Results

As a preliminary evaluation, we ran CoRD on two real concurrent programs: the SQLite [26] embedded database (used in Firefox, iOS, Chrome and Android) and Pbzip2 [7] a parallel implementation of the popular bzip2 file compressor. SQLite has around 100 KLOC and Pbzip2 has around 2 KLOC. All experiments were run on a 800 MHz 48 core AMD Opteron 6176 machine with 512 GB of RAM running Ubuntu Linux 11.04 with kernel version 2.6.38-13. All time-related results represent averages over 10 experiments.

4.1 Effectiveness of detection

CoRD first statically detected 88 potential races for SQLite and 122 races for Pbzip2. The static race detection took 55.57 seconds for SQLite and 0.52 seconds for Pbzip2, on average over the 10 runs.

Table 1 shows the results of data race detection.

Program	**SQLite**	**Pbzip2**
Potential races	88	122
Crashes	0	4
Deadlocks	3	0
No observable effect	0	1
Likely false positives	37	63
Not encountered	48	54

Table 1. Data race detection with CoRD. The *"Potential races"* refers to races the races reported by the static race detector. The next three categories correspond to true positives (3.3.1):*Crashes* and *Deadlocks* refer to crashes and deadlocks that occur in one of the primary or alternate interleavings. Races in the *"No observable effect"* category represent true races for which the execution of the primary and alternate did not cause any observable program failure (crash deadlock or fired assertion, in our case).*"Likely false positives"* were explained in (3.3.2). *"Not encountered"* races correspond to potential races that were not encountered in our test runs.

In order to perform validation, we used a single test case for SQLite that initializes the database and inserts some items to it and subsequently removes them. We used two test cases for Pbzip2: one that does compression and another one that does decompression. CoRD managed to uncover 3 races that cause a deadlock in SQLite, 4

races that cause a crash in Pbzip2, and 1 race that did not have any observable effect. Prior work has reported 1 of the races that cause a deadlock, 3 of the races that caused a crash and the race with no observable effect [14]. To our knowledge, the other races were not previously reported.

The large number of "*Likely false positives"* confirms prior work that reports many false positives for static race detection [28]. We believe that the large number of *"Not encountered"* races is due to the limited set of test cases we used for our preliminary evaluation, which limits the number of paths explored through a program.

4.2 Performance Evaluation

To evaluate the runtime performance overhead incurred by CoRD, we first measured the effect of the instrumentation without performing any validation, and compared this with native execution. We observed 0.91% overhead for Pbzip2 and 0.99% overhead for SQLite.

Next, we enabled validation and measured the overhead of CoRD for Pbzip2 and SQLite by only validating a single race at a time. This case incurred at most 1.6% overhead.

4.3 Efficiency Comparison to Other Detectors

For comparison, we performed pure dynamic race detection for Pbzip2 and SQLite using Google's ThreadSanitizer [16] while running the same test cases used by CoRD. The average overhead ranged from $3,001\%$ for Pbzip2 to 972% for SQLite with the peak overhead being $4,200\%$. Note that we give ThreadSanitizer the benefit of access to all executions that CoRD has access to. This is probably overly generous, because dynamic race detection is not crowd-sourced to multiple users. So one would run ThreadSanitizer on fewer executions and obtain lower coverage than shown here.

If we had performed pure static race detection, we would have obtained the same total number of races for Pbzip2 and SQLite. However, the classification in Table 1 would not have been available, therefore developers would not have information on how to prioritize the fixing of races. This would in turn impact the users, because it might take longer to remove the data races with severe consequences. The benefit of tolerating the $< 2\%$ overhead of CoRD is that race detection results are more detailed and helpful.

4.4 Bug manifestation probability

Finally we evaluated how CoRD increases the probability that concurrency bugs manifest themselves. For this, we ran the SQLite test case 10,000 times and never encountered any deadlock. In contrast, CoRD caused the occurrence of 3 deadlocks from a maximum of 176 executions.

5 Related Work

CoRD is inspired in part by Windows Error Reporting (WER) [8], a large collaborative error reporting system. WER collects information (memory dumps, call stacks, etc.) after a crash, in order to prioritize potential bugs. WER does not formulate any hypotheses regarding a potential bug before encountering it. WER crowd sources program executions and gathers reports after crashes or hangs. Then, it formulates the hypothesis about a potential bug. This hypothesis has to be validated manually.

CoRD reverses this process: it formulates the hypothesis prior to crowd sourcing (static race detection) and then uses crowd sourcing for validation tasks instead of executions. Moreover, in CoRD, validation is entirely automatic. Having a hypothesis together with automated validation can potentially be more valuable: it will save the developer the time-consuming detective work of connecting the hypothesis to the validation.

CoRD's distributed and collaborative approach is in part influenced by statistical bug isolation [19], which transforms an assertion-dense program into several programs with fewer assertions. Users execute these programs and report back the triggered assertions. Similarly, AjaxScope [15] enables error-reporting, performance profiling, and leak detection. AjaxScope uses on-the-fly instrumentation of JavaScript code to distribute the overhead across many users. CoRD distributes race validation tasks among its participants, and increases the chances of manifestation of data race-related failures.

Kivati [4] is a system that statically detects potential atomicity violations and avoids the potential violations at runtime. Kivati requires source code instrumentation and a modified kernel. It's runtime overhead can go up to 35%. Similarly to Kivati, CoRD uses static analysis to detect potential races. CoRD validates data races in a distributed manner to reduce runtime overhead and does not require a custom kernel. Moreover, CoRD can learn from failing executions and has the potential to avoid only known-to-be-harmful interleavings.

CTrigger [24] is a system that takes as input a trace of program execution and performs an analysis of this trace to identify accesses that are potentially involved in an atomicity violation. Subsequently, it re-executes programs and injects delays between instructions to increase the chances of triggering an atomicity violation. CTrigger requires a recording infrastructure and performs offline replay and analysis. Similarly, CoRD increases the chances to encounter race-related problems. However, CoRD does not rely on any execution recording and has low runtime overhead.

6 Future Work

CoRD's race detection does not rely on alias analysis, and this potentially increases both the number of false positives and false negatives in race detection. We plan to use alias analysis to perform more accurate static race detection.

We plan to explore a solution which involves distributing "fixes" back to the pods. It is possible to avoid cer-

tain interleavings in subsequent runs of the instrumented binary by distributing a list of to-be-avoided schedules, allowing the software to gain immunity against data race bugs. CoRD can be used to derive the execution filters used by LOOM [29] to avoid races at runtime.

Currently the assignment of potential data races to pods is done randomly. This is inefficient, because the hive may enable monitoring for a potential data race for a user that never executes the racing memory accesses. In future work, we intend to improve this by leveraging the path sampling statistics gathered by the pods: each pod can monitor which racing accesses are executed locally and forward this information to the hive. The hive can then make more informed assignment decisions of data race reports to pods.

We are currently working on better ways to generate the timeout values used for thread schedule orchestration. One option is to derive the value of the timeout using static analysis and estimating the time required for the second racing access to occur [17]. Another option is to try out different timeout values on different pods, so that only few users are impacted by larger timeouts.

The general concept of a distributed collaborative framework has privacy implications, although in the particular example of data race detection, the information that is gathered from the pods is minimal. We are considering general ways to quantify the balance between privacy and the amount of execution information that is relayed from the pods to the hive.

Finally, we plan to extend the idea of collaborative information recycling to other types of bugs. We believe that, with sufficiently specialized instrumentation, other types of bugs can also be efficiently triggered.

7 Conclusion

We propose CoRD, a collaborative framework that statically detects data races and then dynamically validates them by leveraging execution information from all its participants. Our initial results show that CoRD is effective in detecting races and has low overhead, while increasing the chances of manifestation of race-related failures which enables better testing and validation.

Acknowledgments

We thank Coverity for their support regarding their static analysis tool.

References

[1] A history of Windows. `http://windows.microsoft.com/en-us/windows/history`.

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

[3] G. Candea. Exterminating bugs via collective information recycling. In *HOTDEP*, 2011.

[4] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

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

[6] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[7] J. Gilchrist. Parallel BZIP2. `http://compression.ca/pbzip2`.

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

[9] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[10] Google. Google chrome blog. `http://chrome.blogspot.ch/2012_06_01_archive.html`.

[11] Intel Corp. Parallel Inspector. `http://software.intel.com/en-us/articles/intel-parallel-inspector`.

[12] James Pearn. How many servers does google have? `https://plus.google.com/114250946512808775436/posts/VaQu9sNxJuY`.

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

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

[15] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Symp. on Operating Systems Principles*, 2007.

[16] T. I. Konstantin Serebryany. ThreadSanitizer - data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[17] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, 2012.

[18] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, July 1993.

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

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

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

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

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

[24] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

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

[26] SQLite. `http://www.sqlite.org/`, 2010.

[27] The Associated Press. General Electric acknowledges Northeastern blackout bug. `http://www.securityfocus.com/news/8032`.

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

[29] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Symp. on Operating Sys. Design and Implem.*, 2010.

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