Failure Sketches: A Better Way to Debug

Baris Kasikci[1] Cristiano Pereira[2] Gilles Pokam[2]
Benjamin Schubert[1] Madanlal Musuvathi[3] George Candea[1]
[1] School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL)
[2] Intel Corporation
[3] Microsoft Research

Abstract

One of the main reasons debugging is hard and time consuming is that existing debugging tools do not provide an explanation for the root causes of failures. Additionally, existing techniques either rely on expensive runtime recording or assume existence of a given program input that reliably reproduces the failure, which makes them hard to apply in production scenarios. Consequently, developers spend precious time chasing elusive bugs, resulting in productivity loss.

We propose a new debugging technique, called *failure sketching*, that provides the developer with a high-level explanation for the root cause of a failure. A failure sketch achieves this goal because: 1) it only contains program statements that cause a failure; 2) it shows which program properties differ between failing and successful executions. We argue that failure sketches can be built by combining in-house static analysis and crowdsourced dynamic analysis. For building a failure sketch, we do not assume that developers can reproduce the failure. We show preliminary evidence that failure sketches can significantly improve programmer productivity.

1 Introduction

Debugging is the process of eliminating bugs in a program. Traditional debugging is a cyclic process, that is, it involves running a failing program in a debugger over and over again, hoping to reproduce the failure, learn its root cause and eventually fix it. Debugging takes a significant developer time (around 50% [19]), because it requires deep understanding of the program code.

A problem with traditional debugging is that it may not be possible to reproduce the failure. This is also a problem for automated techniques like delta debugging [27], which relies on repeatedly reproducing a failure for the purpose of isolating program states that cause the failure.

To solve the bug reproduction problem, record/replay systems have been proposed [4, 20] to record failing executions and deterministically replay them. Record/replay systems can be helpful, however, despite many efforts, they have not seen widespread adoption. This is because these systems still have prohibitive overheads for production use, especially on multiprocessors (e.g., 400% for SMP-ReVirt [8]). State-of-the-practice record/replay systems incur high overhead for parallel programs, as they serialize executions by emulating a single-core machine [2, 3]. Therefore, reproduction of software bugs through record/replay may not be feasible in general. In the absence of a replayable execution, developers may not be able to reproduce the failure to debug a program.

Furthermore, debugging requires root cause diagnosis, and merely reproducing a failure (e.g., through record/replay) does not solve this problem. A root cause is a cause or combination of causes, that once removed from the program, prevents the failure associated with the bug to recur [26]. Record/replay does not solve the root cause diagnosis problem, because the developer still needs to grok what distinguishes a failing execution from a successful execution to understand the root cause of a failure. This can be difficult if the program is complex, has tightly coupled modules, and the failure has complex data and control dependencies. This complexity is exacerbated if the replayed execution contains a lot of information that is not relevant to the failure (e.g., control and data flow information that does not affect the failure).

Guided by these observations, we posit that a new paradigm for debugging is necessary, because neither traditional debugging nor record/replay debugging nor delta debugging, as provided by today's tools, solve the root cause diagnosis problem, which is at the heart of debugging programs. We argue that, to debug programs effectively, a developer needs to:

1. Have access to a partial execution trace that we call the *failure sketch*; the ideal failure sketch is composed of the relevant data and control flow information that allows reasoning about the failure, but nothing else unrelated to the failure. The advantage of the failure sketch over a full execution trace is that it does not contain superfluous execution information: all of its elements pertain to the failure.
2. Determine the differences in relevant control and data flow information between failing and successful program executions by using the failure sketch.

We argue that the process of obtaining failure sketches should reflect how failures occur in the real world. We do not assume that we can reproduce the failures (e.g.,

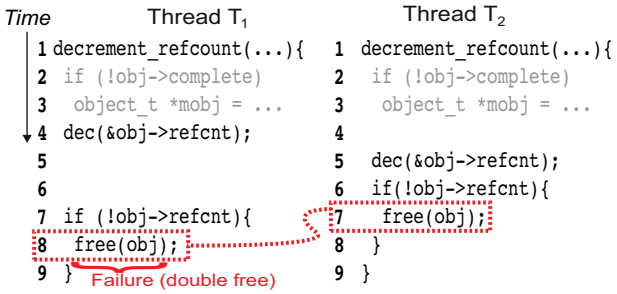

Figure 1: The failure sketch of Apache bug 21287.

through record/replay), because this assumption may not hold in a real setup. We can build failure sketches with as few as a single failure; and we simply need the failures to recur a few times (2-3) to build an ideal failure sketch.

We now explain in detail what failure sketches are and how they can be used for debugging (§2), the challenges of building failure sketches (§3), how to obtain failure sketches automatically (§4), some preliminary results (§5), and related work (§6).

2 What are Failure Sketches and How to Use Them?

An ideal failure sketch is an ordered, per-thread sequence of program statements, which clearly shows the differences between failing and successful runs in terms of data and control flow. If some elements of an ideal failure sketch are missing and/or inaccurate, we call the resulting failure sketch an imperfect failure sketch.

Fig. 1 shows the failure sketch of Apache bug 21287 [1]. Time flows downward in the vertical direction and the steps in the execution are enumerated along the flow of time. The failure occurs when threads T_1 and T_2 execute the `decrement_refcount` function concurrently, which frees `obj` when `obj->refcnt` is 0. Because the check `if(!obj->refcnt)` and the freeing are not done atomically, `obj` may end up being freed twice. The failure sketch shows the program statements that leads to the failure. The dashed line shows the property of the failing execution that differs from successful executions, i.e., `obj->refcnt` getting freed twice.

Failure sketches allow the developer to focus on what is essential for understanding the root cause. They achieve this by concisely displaying the sequence of instructions that lead to the failure. All the statements with black font in Fig. 1 influence the outcome of the failure, and these statements also happen to be the only ones that influence the failure. We grayed out some surrounding statements that do not influence the failure.

Failure sketches also allow developers to zoom in on the elements of the execution that differentiate failing and successful executions, thereby allowing them to identify root causes of failures. Failure sketches achieve this by clearly highlighting such differences (e.g., the dashed line in Fig. 1). These differences point to the root causes of bugs as per the definition of the root cause that we gave in §1: If such differences were to be eliminated, the failure would disappear in most cases. Only if an unrelated root cause results in the same failure, we would need to eliminate other differences from another failure sketch.

We believe that failure sketches provide what developers need to debug programs effectively: a concise execution trace composed of the differences of relevant control and data flow information between successful and failing runs, which allows diagnosing failure root causes. Merely reading the sketch allows a developer to see the program statements that are involved in the failing run and the failure root cause. For the example in Fig. 1, the developer simply needs to ensure that the check `if(!obj->refcnt)` and the freeing of `obj` occur atomically.

3 Challenges

Automatically and efficiently building failure sketches comes with a number of challenges. In this section, we list three main challenges, and in the next section, we discuss ways in which we believe these challenges can be addressed.

Challenge #1: Not only developers may not be able to reproduce failures that occur in the real world, but such failures may not recur at user sites, or they may recur so infrequently that it becomes hard to gather enough data from failing executions to build failure sketches. We argue that even for failures that occur once, we should be able to build a failure sketch, albeit an imperfect one.

Challenge #2: It is difficult to ensure that failure sketches are concise and accurate for complex programs. Developers have limited debugging time, therefore failure sketches should correctly provide the essential information that pertains to a failure.

Challenge #3: It is difficult to make the construction of failure sketches fast, while not imposing large runtime overheads and significantly perturbing real-user executions. However, this is essential, because a high-overhead solution would not be practically applicable in the real world, and perturbing real user executions a lot may mask the failures.

4 Obtaining Failure Sketches

At a high level, failure sketching takes as input a program and the failure and outputs the failure sketch. We do not envision to do bug detection as part of failure sketching. In other words, we assume that the failure is provided

to failure sketching via a bug report, core dump, or a program statement where the failure manifests itself.

Failure sketching aims to strike the right balance between static and dynamic analysis to be able to always build a failure sketch that is meaningful for the developer, while not incurring prohibitive runtime overhead. We argue that leveraging the right technologies will solve the challenges we mentioned in the previous section.

Static analysis is done offline, so it does not incur any runtime overhead. We envision static analysis to use various forms of failure reports (e.g., a coredump). Using these, static analysis can extract the failure point. Then, it can compute a *slice* [25] of the program, which is a group of program statements that have data and control dependencies to the failure point. Slice computation can leverage information such as the call stacks that is present in the coredump to increase the accuracy of slices.

The static slice will contain some statements that do not affect the failure, because it is computed offline. To mitigate this, we propose refining the static slice by gathering data from real-world executions. Slice refinement removes from the static slice the components of the slice that do not appear in real executions.

To refine the slices, we propose a crowdsourced scheme that relies on always-on hardware support to collect traces from real-world executions. Modern processors offer hardware tracing features to monitor the execution of software [5, 10]. Hardware support allows capturing detailed traces with low runtime overhead, making it suitable for user-site deployment. In the next section, we show that our proof of concept uses Intel Processor Trace (Intel PT) [10] traces that are gathered from real-world executions to refine the slices, but any other hardware tracing mechanism containing relevant information to can be used, such as thread ordering [21]. Intel PT traces contain per-thread control flow traces and provide a per-thread correct ordering of control flow events.

We can solve challenge #1, namely building failure sketches when failures do not recur, with a combination of static analysis and crowdsourced always-on hardware tracing. The first time a failure occurs, we can compute a static slice and refine it using hardware traces. Such a slice is an imperfect failure sketch: it neither has the correct ordering of program statements as they occurred in the actual execution nor does it show all the differences between failing and successful executions.

Challenge #2, namely building accurate and concise failure sketches, can be partially solved by further user-site tracing. We propose tracking the data flow in addition to the control flow, as some failures depend on the data flow. To track the data flow, we can instrument the memory accesses in the slice, using the watchpoint support present in modern processors. Watchpoints allow breaking when a read from or a write to a certain address occurs. Moreover, watchpoints can also allow tracking the order of memory accesses across cores, which is a necessary information for reasoning about concurrency bugs. If the refined slice contains more memory locations than the available watchpoints on a user machine, we can resort to sampling to track the memory accesses on several users' machines [12, 18]. If available, failure sketching could use hardware tracing features to capture values of memory operations instead of watchpoints.

To entirely solve challenge #2, we need to perform user-site tracing for successful executions as well. This way, we can identify the differences of program properties between failing and successful runs. We argue that these differences point to the root cause by observing that the programmer patches fixing the bugs eliminate these differences, similar to how prior work evaluates the accuracy of root cause diagnosis [6, 22]. We present an example that supports our claim in the next section.

Challenge #3, namely building failure sketches efficiently, would be solved if we solve the previous two challenges using low-overhead hardware support. Intel Corporation indicates that hardware-based control flow tracing, the feature of Intel PT that we rely on, targets performance overhead to be lower than 5% [7]. We believe that overheads of 5% are low enough for always-on tracing, but we are working on selective recording techniques to further lower this overhead. Through this exercise, we hope to identify the ideal filtering mechanisms to minimize the amount of information collected.

5 Preliminary Results

5.1 Prototype

We implemented a prototype to show that it is feasible to solve the challenges of building failure sketches.

For static analysis we relied on LLVM [15], which provides a lot of analyses out of the box. We relied on modules of the LLVM framework that allow building the control flow graph of the entire program and have support for alias analysis [14].

For control flow tracing, we implemented a PIN-based [17] simulator of Intel PT. The driver support for the real hardware is being made available by Intel at the time of this writing. Our simulator relies on PIN's binary instrumentation to simulate Intel PT support, and therefore it has high runtime overhead (around 10×). We did not optimize the simulator's performance, because we intend to replace it by the actual hardware in the future.

We used source code instrumentation for supporting watchpoints, which incurs no perceptible runtime overhead, which is promising. Nevertheless, we are investigating new hardware mechanisms to efficiently collect more data flow information than what watchpoints provide.

```
   cons(queue* f){
889 mutex_lock(f->mut);
... ...
... if (allDone == 1) {
898  mutex_unlock(f->mut);
    }
   ...
```

(a)

```
#0 in cons at pbzip2.cpp:898
#1 start at pthread_create.c:312
#2 in clone at ...clone.S:111
```

(b)

→ Failure, line 898 (segfault)

Figure 2: The failure in pbzip2 (a), the stack trace (b)

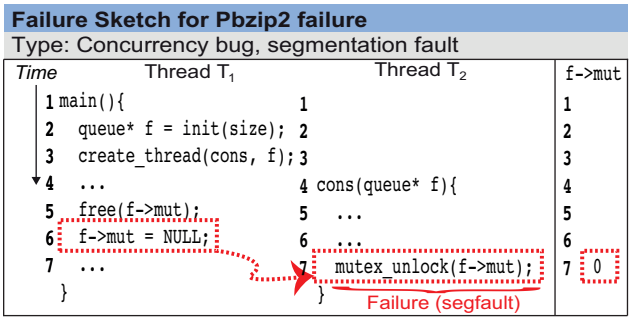

Figure 3: The failure sketch of the pbzip2 bug

Failure sketching instruments programs running on users' machines to gather data flow information, which may raise intrusiveness and privacy violation concerns. To be less intrusive, we can dynamically instrument the programs using dynamic binary rewriting. To reduce privacy concerns, we can quantify the amount of data that the instrumentation can leak, and try to minimize that.

5.2 Pbzip2 Concurrency Bug

For the sake of simplicity, we picked a small C++ program (2 KLOCs), namely pbzip2 [9] which is the multithreaded version of the file compression program bzip2. The bug in pbzip2 is a concurrency bug, which is a class of bugs known to be difficult to debug [13, 28]: it occurs only under a specific thread interleaving that causes the control flow of the execution to change, which in turn changes the data flow of the program, causing the program to fail due to a segmentation fault.

Fig. 2.(a) shows a code snippet from pbzip2. When the program fails, the coredump contains the stack trace shown in Fig 2.(b). The failure occurs in the `cons` function on line 898 because `f->mut` is `NULL`.

5.3 Manual Debugging

We describe our manual debugging effort to later contrast it to debugging with failure sketches. When we encountered the failure on line 898, we realized that `f->mut` is also accessed on line 889, however the program had not crashed. We concluded that `f->mut` was modified by another part of the program before it was accessed on line 898. Alas, we could not determine which part of the code was modifying `f->mut` by reading the code.

Based on our only lead that `f->mut` was being modified somewhere in the program that we could not determine, we thought of using watchpoints. However, there was one caveat: one needs to know the address to watch in order to place the watchpoint, and this can only be known at runtime. Therefore, we put a regular breakpoint on the `cons` function in Fig. 2 on line 889 where we knew that `f->mut` will have been allocated an address. When this breakpoint got triggered, we placed a watchpoint at the address of `f->mut`. We also wrote scripts that print the program counter when this watchpoint would be triggered. We continued executing the program, but the bug wouldn't recur. We realized that we are dealing with a Heisenbug.

We replaced the breakpoint on `cons` with a tracepoint, which is similar to a breakpoint, but allows automatically running a sequence of commands when hit. Tracepoints stop the execution less than breakpoints, thus they perturb timing less. We wrote commands that would place a watchpoint at the address of `f->mut` when the tracepoint in `cons` is hit. Then, we ran pbzip2 10,000 times with 10 different inputs that we selected for input diversity. We managed to reproduce the bug only twice. Further investigation of the logs containing the program counters revealed that the root cause of the bug was another part of the program deallocating `f->mut` and setting it to `NULL` while the program executed the statements between the lines 889 and 898. This entire process took us 20 hours.

5.4 Construction of a Failure Sketch

Failure sketching automates the manual root cause diagnosis process for the pbzip2 failure, and it builds the failure sketch in Fig. 3. The failure sketch shows that `init` allocates memory for `f` (step 2). It is not shown for brevity, but `init` also allocates memory for `f->mut`. Then, `main` creates a thread with the start routine `cons` and the argument `f` (step 4); T_2 starts executing, but T_1 gets scheduled (step 5), and it frees `f->mut` and sets it to `NULL` (steps 5-6). When T_2 accesses `f->mut` (which aliases to `f->mut`), it crashes (step 8) because of the particular scheduling of threads illustrated by the dashed arrow. We now explain in detail how failure sketching automatically computes the failure sketch for this bug.

When the failure in pbzip2 first occurs, failure sketching automatically identifies the refined slice using its static slicing algorithm and Intel PT traces. We assume that user sites have always-on Intel PT tracing, therefore the Intel PT trace for this bug is available the first time the failure occurs. The refined slice of the pbzip2 bug is composed of the statements in Fig. 3, and a few other statements not shown in Fig. 3, notably the statements in `init` that allocate the memory for `f->mut`.

Failure sketching identifies the refined slice using a

single failing execution. The refined slice is imperfect: it does not order statements across threads and does not show differences between failing and successful executions, but it identifies the statements that are involved in the failure separately for each thread. We believe that the imperfect failure sketch eases the root cause diagnosis problem for the developer. For the pbzip2 example, it is not difficult to guess that the statement `f->mut = NULL` precedes the statement `mutex_unlock(f->mut)` and this order of accesses results in a segmentation fault.

Nevertheless, failure sketching can use more user executions to build the ideal failure sketch automatically, without resorting to any guesswork. For this, failure sketching places a watchpoint to the address of `f->mut` at user sites that run pbzip2, whenever `f->mut` is allocated in `init`. Failure sketching monitors the triggering of this watchpoint at runtime and logs the accessing thread id as well as the program counter of the access.

When the failure recurs, watchpoint-based tracking allows failure sketching to determine that T_1 executes the statements in steps 5 and 6, followed by T_2, which executes the statement in step 7. This tracking allows failure sketching to order the statements across threads in a failing execution as shown in Fig. 3.

Failure sketching tracks multiple user executions to identify the differences between failing and successful runs with respect to several properties of the refined slice (i.e., computed values and flow of instructions). For pbzip2, failure sketching will determine that, in failing executions, T_1 will free `f->mut` and set it to `NULL` followed by the dereference of `f->mut` in T_2. This difference is captured using the dashed arrow in Fig. 3. Successful executions will not exhibit this ordering behavior.

5.5 Using a Failure Sketch

Failure sketching automates the process of generating the failure sketch for the pbzip2 failure by combining static and dynamic analysis. The developer can trivially use the failure sketch to fix the bug. For the failure in pbzip2, the developer needs to introduce proper synchronization that will eliminate the offending thread schedule. This is exactly how pbzip2 developers fixed this bug [9].

Manually debugging the pbzip2 failure took us 20 hours, whereas failure sketching automatically builds the failure sketch after having witnessed 2 failing executions and a successful execution in 2 seconds. This represents over four orders of magnitude improvement in root cause diagnosis time.

6 Related Work

Delta debugging isolates the cause-effect chain of a failure by systematically narrowing down the state difference between a failing run and a passing run. Delta debugging requires the failure to be reproducible. Failure sketching does not assume that failures are reproducible and aims to build an imperfect sketch even with a single failing execution. If failures recur at the user site, failure sketching can build more accurate sketches.

Triage [23], Giri [22], and DrDebug [24] use dynamic slicing for root cause diagnosis. Triage uses custom checkpointing support. DrDebug and Giri assume that failures can be reproduced by record/replay and knowledge of failing inputs, respectively. Failure sketching relies on hardware tracing for slice refinement and does not assume that failures can be reproduced.

Crowdsourcing for building failure sketches is inspired by the collaborative bug isolation approaches CBI [16] and CCI [11]. CBI and CCI instrument programs to sample certain predicates (e.g., branch targets) from user executions. Because they use sampling, CBI and CCI require many successful and failing runs to correlate predicates and failures. Failure sketches can be built with as few as a single failing execution.

LBRA/LCRA [6] relies on the last branch record of Intel processors and a hardware extension that allows correlating branches with sequential bugs and coherency events with concurrency bugs, respectively. LBRA/LCRA targets failures due to control flow. Failure sketching targets failures due to either control or data flow, or both. LBRA/LCRA works well for bugs with short root cause to failure distances, whereas failure sketch sizes are limited by persistent storage size.

7 Conclusion

We argued that we need a new debugging paradigm allowing developers do root cause diagnosis using a representation of failures called failure sketches. Failure sketching does not assume that developers can reproduce failures. Failure sketches only contain information pertaining to a failure, and they show differences between failing and successful runs to point developers to the root cause. We argued that it is possible to do this automatically by building a prototype that uses a combination of static analysis and crowdsourced dynamic analysis that relies on modern hardware support. We presented initial results that show significant improvements of programmer productivity. In future, we envision using failure sketches for automated test case generation and for improving the performance of program analysis techniques like symbolic execution.

Acknowledgments

We are indebted to the anonymous reviewers, and to Edouard Bugnion and Emery Berger for their insightful feedback and generous help in improving this paper. This work was supported in part by ERC Starting Grant No. 278656 and by gifts from Intel and VMware.

References

[1] Httpd bug 21287. `http://bit.ly/14BUPN6`.

[2] Mozilla rr. http://rr-project.org/.

[3] UndoDB. http://undo-software.com.

[4] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore programs. In *Symp. on Operating Systems Principles*, 2009.

[5] ARM Corporation. Arm coresight. http://bit.ly/17qNaCQ, 2014.

[6] J. Arulraj, G. Jin, and S. Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[7] Beeman Strong. Debug and fine-grain profiling with intel processor trace. http://bit.ly/1xMYbIC, 2014.

[8] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments*, 2008.

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

[10] Intel Corporation. Intel processor trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

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

[12] 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.

[13] 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.

[14] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

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

[16] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[18] 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.

[19] S. McConnell. *Code Complete*. Microsoft Press, 2004.

[20] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. 2008.

[21] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 216–225, New York, NY, USA, 2011. ACM.

[22] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. 2013.

[23] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *Symp. on Operating Systems Principles*, 2007.

[24] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *CGO*, 2014.

[25] M. Weiser. Program slicing. In *Intl. Conf. on Software Engineering*, 1981.

[26] P. F. Wilson, L. D. Dell, and G. F. Anderson. *Root Cause Analysis : A Tool for Total Quality Management*. American Society for Quality, 1993.

[27] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002.

[28] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.