Automatic On-line Failure Diagnosis at the End-User Site

Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos and Yuanyuan Zhou
Department of Computer Science, University of Illinois at Urbana Champaign, IL 61801

Abstract

Production run software failures cause endless grief to end-users, and endless challenges to programmers as they commonly have incomplete information about the bug, facing great hurdles to reproduce it. Users are often unable or unwilling to provide diagnostic information due to technical challenges and privacy concerns; even if the information is available, failure analysis is time-consuming.

We propose performing initial diagnosis *automatically* and *at the end user's site*. The moment of failure is a valuable commodity programmers strive to reproduce—leveraging it directly reduces diagnosis effort while simultaneously addressing privacy concerns.

Additionally, we propose a *failure diagnosis protocol*. So far as we know, this is the first such automatic protocol proposed for on-line diagnosis. By mimicking the steps a human programmer follows dissecting a failure, we deduce important failure information. Beyond *on-line* use, this can also reduce the effort of in-house testing.

We implement some of these ideas. Using lightweight checkpoint and rollback techniques and dynamic, runtime software analysis tools, we initiate the automatic diagnosis of several bugs. Our preliminary results show that automatic diagnosis can efficiently and accurately find likely root causes and fault propagation chains. Further, normal execution overhead is only 2%.

1 Introduction

Software failures contribute greatly to down time and security holes. Despite expending enormous effort in software testing *prior* to release, software failures still occur during *production runs*, as faults slip through even strict testing. The ubiquity of bugs in production runs is strong testimony to the fact that addressing *production run* software failures is critical.

While much work has been conducted on software failure diagnosis, previous work focuses on *off-line* diagnosis. Tools like interactive debuggers, or more advanced tools like data slicing [16], backtracking [7] and deterministic replay [8, 12] can help, but require *manual* effort, and so cannot be used for production runs. Other *off-line* techniques such as delta debugging [15] and failure-inducing-chop inference [4] automate *some* parts of *off-line* diagnosis, yet impose high overhead.

Current on-line tools focus on data collection, as failure reproduction is difficult. Tools like [1, 8, 13] help, but can slow down programs significantly (over 100X [9]). Reducing overhead requires either sacrificing detail (e.g. by only tracing system calls [14]) or expensive, undeployed hardware extensions [8, 13]. Regardless, users don't release such information due to privacy and confidentiality concerns, and even if provided, such "raw" data is time consuming to manually search for root causes.

It would be desirable if diagnosis was automatically conducted *on-line* as a software failure occurs, and if only summarized diagnosis results were sent back. This has several advantages:

(1) **Leverage the failure environment and state**. Since such diagnosis is done right after a software failure, almost all program states (memory, file, etc.) and execution environment information (e.g. user inputs, thread scheduling, etc.) are available, trivializing failure reproduction.

(2) **Reduce programmer effort**. Similar to the triage system of medicine, software failure diagnosis could *automate* finding some useful clues (e.g. failure-triggering inputs). This would significantly reduce programmer's effort. Of course, nobody expects paramedics to cure patients, nor software to fix bugs, but both can offload the initial steps. Moreover, *such automated diagnosis also applies to* ***in-house testing***.

(3) **Address users' privacy concerns**. If failure diagnosis is conducted *on-line* at the user's machine, private data and execution traces need not be sent out. After diagnosis, results (e.g. buggy instructions, root cause candidates, triggering inputs, etc.) can be sent out in a form more condensed and transparent to users than core dumps or traces. Indeed, we feel that excluding core dumps increases the amount of information programmer's receive: if users are wary of core dumps, the programmer will receive nothing.

(4) **Guide failure recovery and avoid future failures**. Immediate and automatic diagnostic results would be available in time to help failure recovery in subsequent executions. For example, identifying failure-triggering inputs allows such inputs to be filtered *prior to a programmer creating a patch*.

Although on-line diagnosis is clearly desirable, it faces

several challenges. First, normal execution overhead must be low; users are unwilling to sacrifice much performance for reliability. Second, on-line diagnosis mandates a lack of human guidance. All information collection, reasoning, and analysis must be automated. Finally, diagnosis times must be short. End users are impatient, and even if normal-run overhead is low, diagnosis time is strictly limited.

1.1 Our Contributions

In this paper, we propose a system (the first, to the best of our knowledge), for *automatic on-line* diagnosis of software failures that occur during *production runs*. Specifically, we address the above three challenges with two innovative approaches:

(1) Capture the failure point and conduct *just-in-time* failure diagnosis with checkpoint-re-execution system support. Traditional techniques expend equal monitoring and tracing effort throughout execution; this is wasteful given that most production runs are failure-free. Instead, by taking light-weight checkpoints during execution and rolling back after a failure, diagnosis must be done only *after* a failure has occurred. Heavy-weight code instrumentation and advanced analysis can be repeatedly applied post-failure re-execution, minimizing normal-case overhead. In this scheme, the diagnostic tools have most, if not all, failure-related information at hand, while diagnosis is focused on the execution period closest to the failure. ***In combination with system support for re-execution, heavy-weight tools become feasible for on-line diagnosis.***

(2) A novel automated top-down human-like software failure diagnosis protocol. We propose a framework (see Figure 1 which assembles the manual debugging process. This Triage Diagnosis Protocol (TDP) takes over the role of humans in diagnosis. At each step, known information is used to select the appropriate next diagnostic technique. Within the framework, many different diagnosis techniques, such as core dump analysis, bug detection, backward slicing, etc., may be integrated. Additionally, ***using the results of past steps to guide and as inputs to future steps increases their power and usefulness.*** In combination, these tools can potentially extract and deduce much useful information about a software failure, including bug type and location, likely root causes and fault propagation chains, failure triggering inputs, failure triggering execution environments, and even provide potential temporary fixes (such as input filters).

2 Idea

The goal is to perform an initial diagnosis before reporting back to the programmers. This is similar to the triage system used in medical practice: in case of emergency, paramedics must perform the first steps: providing first aid, sorting out which patients are most critical, measuring vitals, etc. Similarly, given a software emergency (a program failure), programmers cannot immediately begin debugging. The immediate response must be provided by some pre-diagnosis tools to provide the programmer with useful clues for diagnosis.

To realize this goal, in light of the previously mentioned challenges, we innovatively combine several key, novel ideas:

(1) Capture the moment of failure. When a program fails, we have something valuable: the failure state and failure-triggering environment. Rather than let this moment pass, we capture it and leverage it fully. Therefore, unlike most previous approaches that pay high overhead to record copious information (e.g. execution traces) to *reproduce* the failure off-line, we propose to perform diagnosis *immediately* after the failure.

(2) Exploit sandboxed re-execution to reduce normal-run overhead. Rather than instrumenting continuously, system support for rollback and re-execution from a recent checkpoint allows analysis only *after* a failure has occurred. This eliminates the "*blind*" collection of information during normal runs to greatly reduce overhead–instead this is deferred to after a failure. Also, failure symptoms and the results of previous analysis can allow a *dynamic* determination of what needs to be done. If we already have evidence that the bug is probably deterministic, a race detector is useless–without rollback the instrumentation must be "always on" even for unrelated bugs.

(3) Automatically perform human-like top-down failure diagnosis. Inspired by manual debugging, the TDP provides a top-down failure-guided diagnosis. Based on the failure symptom and previous diagnosis results, TDP automatically decides the next step of instrumentation or analysis.

(4) Leverage various dynamic bug detection and fault localization techniques. The TDP, in combination with re-execution, allows many existing (or future) bug detection and fault isolation techniques to be applied. As proof of concept, we prototype a memory bug detector and a program backward slicer, both of which are dynamically plugged in during re-execution via dynamic binary instrumentation. Although information from the very beginning of execution is unavailable, working around this requires only minimal modifications. The memory bug detector synergistically provides input to the backward slicer, resulting in accurate failure information.

2.1 Triage Diagnosis Protocol (TDP)

This section, and Figure 1, describe the human-like top-down Triage Diagnosis Protocol. This is, of course, incomplete. Debugging is something of an art; the TDP

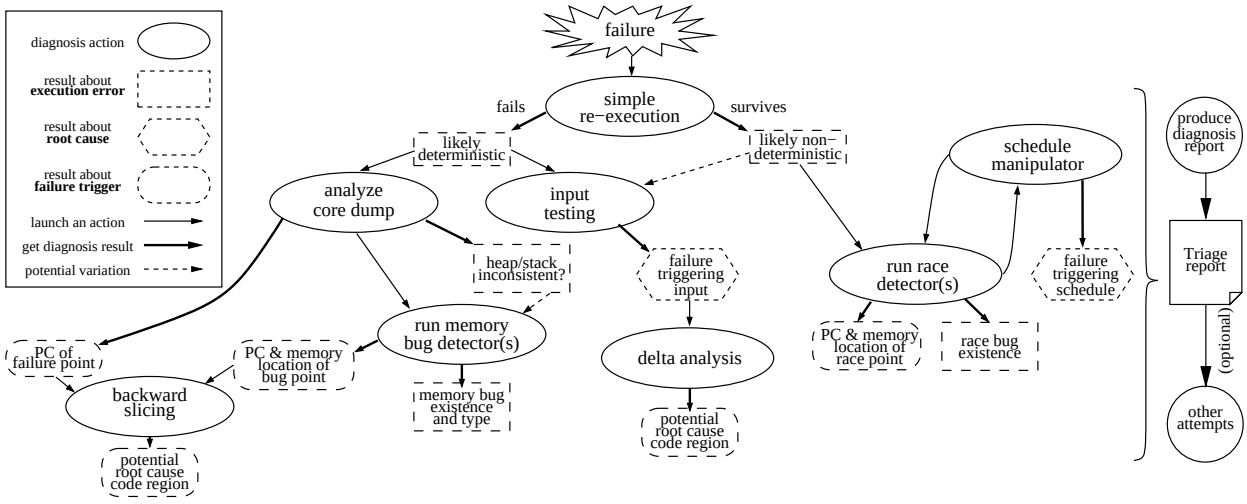


Figure 1: Diagnosis protocol (TDP) diagram (This is a sample illustration of the Triage diagnosis protocol using representative bug detection and diagnosis techniques. Other techniques can easily plug into the Triage protocol and framework.)

is inspired by our own debugging processes, as well as an attempt to order steps such that the results of earlier stages are useful for later stages to consume (e.g. input testing reduces the work of later stages). We expect that further techniques and steps will be added.

Simple replay As our goal is to provide the groundwork for a programmer to start debugging, we mimic the initial step a programmer would follow: simply retry the execution. This trivially distinguishes deterministic and non-deterministic bugs. Since subsequent steps vary drastically depending on the initial classification, this is done first.

Core dump analysis Assuming a deterministic bug, the next step analyzes the memory image at the time of failure. This inexpensive step gives the crashing instruction and related variables–a starting point for further analysis. Also, the consistency of the heap and stack can identify some memory bugs.

Input testing The next step is input testing. First, this identifies the failure-triggering input to avoid overloading the programmer and subsequent TDP steps with extraneous information. Second, this extracts signature of the failure-triggering input. A programmer, given one test case, will likely try to find the minimum input or the particular request that triggers the failure[15]. The TDP also does this, but *automatically* with the support of re-execution. An extra possibility is to build a temporary fix by filtering bad inputs (prior to a patch release).

Dynamic memory bug detection After input testing comes dynamic bug detection. This step is after core dump analysis as it is slower and may be partially skipped if the core dump report is clean. There are many existing powerful bug detection tools programmers use–they are not used in production runs due to the huge overhead (up to 100X slowdowns [17]) potentially involved. However, such tools can be dynamically plugged in during re-execution, when overhead is no longer a major concern. This step catches common bugs such as buffer overflows, data races, double frees, etc. Both positive and negative results are useful for understanding the failure—detecting no buffer overflow can avoid a wild goose chase for non-existent buffer overflows.

Dynamic backward slicing From the previous diagnostic steps, we have a point along the fault-propagation chain: the crashing instruction (from core dump analysis) or a misbehaving instruction (from bug detection). Commonly, a programmer will manually trace backward with break- and watch-points. Dynamic backward slicing automates this process. Previous work [8] shows that fault propagation lengths tend to be short. A backward slice of a faulty execution isolates the portions of the program responsible for the failure, and the initial fault will likely appear quite close to the starting point for the slice.

Delta analyzer The delta analyzer runs after the input tester. If the input tester provides two similar runs, one failing and the other not, the delta analyzer will compare the execution paths and variable values in the two runs. The comparison results can isolate buggy code regions and abnormal variable values.

Non-deterministic bugs If the first step indicates a non-deterministic bug, employing a race detector during the re-execution is the second TDP step. Multi-processor machines are difficult to support for deterministic replay; it may take several exections to repeat conditions sufficiently to trigger a race detector. In uni-processors, however, deterministic replay can easily replicate the race for a race detector to find. After candidate races are identified, re-execution with schedule manipulations can ver-

ify true positives and eliminate false positives. This also gives a candidate thread interleaving to trigger the failure.

Report All of the analysis steps end in producing a report. Different stages can be cross-correlated with each other for ranking purposes, with more precise results (e.g. memory bug detection vs. core dump analysis) prioritized. The summary report gives the programmer a comprehensive analysis of the failure, which is much more useful than an ordinary bug report. Manual bug reports tend to consist of a description of the symptoms, plus some description of how the user triggered the bug; symptoms can be misleading, and actually duplicating a bug from these instructions is often difficult. Previous automatic "Send Error Report" tools send a collection of environmental data plus a core image; this still doesn't allow duplication of the bug, nor does it include any information about what happened prior to the failure. Our proposed reports are composed of *processed* information about the failure, including the execution prior to the failure itself. This gives the programmer information that was previously unavailable, and does not require as much manual inspection to understand.

Speculative techniques In addition to the above analysis, more speculative techniques can also be used. For example, trying to skip potentially problematic code regions or "forcing" variable values may help further isolate bugs.

Temporary fixes A final, optional step, is to attempt to automatically fix the bug. This may be done by automatically filtering bug-triggering inputs (as in [3, 6]) or avoiding bug-triggering schedules or memory layout. If the diagnosis is for a buffer overflow, such objects can be allocated with large paddings (similar to our previous work Rx [10] and others [2, 11]).

3 Experimental Results

In this section, we present preliminary results on a few bugs (Table 1) by combining memory bug detection and backward slicing. We also present performance results showing the feasibility of production-run deployment.

3.1 Diagnosis Results Overview

Table 2 presents the results of combining memory bug detection and backward slicing to diagnose bugs. For all three of the bugs, the specific type of bug (heap-overflow, stack-smashing, or null-pointer) is correctly identified. Especially for BC and NCOMP, this is extra information beyond what a core-dump would provide. Furthermore, backward slicing from the initial fault reaches the root cause instruction in 6 instructions or less.

Memory bug detection As shown in Table 2, the type of bug is correctly determined for all 3 failures. Also, in the case of BC and NCOMP, memory bug detection provides a point along the fault propagation chain much closer to the fault than the actual failure point. Heap and stack overflows corrupt data without causing a failure *at that time*. This points to a shortcoming of core-dumps– the programmer may be able to tell that a heap or a stack overflow occurred, but has no idea of where the overflow happened. Also, for the NCOMP bug, the corruption of the stack prevents simply walking back along the stack activation records to trace the bug. Another benefit is improving the later backward slicing. The more precise point along the fault propagation chain reduces the size of the propagation tree and reduces the time needed.

Fault propagation tree The backward slicer **automatically** generates fault propagation trees. In all cases, the root cause instruction appears in the top 6 candidate instructions. For NCOMP and TAR, the faulting instruction is far from the failure point: for NCOMP, the root cause is 70 source lines away from the stack overflow, and for TAR, the fault is not even in the same source file as the crash point. By tracing the flow of data and control, backward slicing eliminates much extraneous code.

3.2 Detailed bug study: TAR

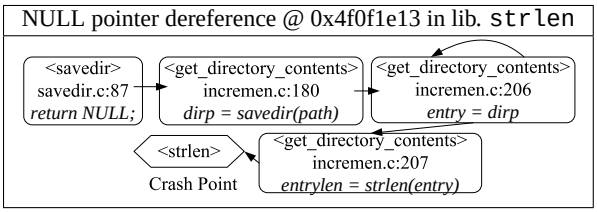


Table 3: Results for tar-1.13.25

As shown in Table 3, the failing instruction is a null pointer dereference by `strlen`. Unfortunately, memory bug detection only gives the failing point, and not why the pointer is null. Backward slicing results balance the lack of information about the bug. As shown in the fault tree of the report, the crashing code is `strlen(entry)`, shown at the root of the tree. Tracing backward along the tree, we will see that the argument `entry` is returned from `savedir`. Additionally, NULL is explicitly assigned to it in that function. Actually, if we look at the source line just above the NULL assignment, we will see that this NULL assignment happens when an `opendir` call to the input directory fails. Note that even though the fault tree is short and simple, it is difficult to get it without backward slicing. It is impossible with just a core dump, because the function `savedir` has already returned and so the stack frame is gone. Also, just looking at the source code is

Name	Program	App Description	#LOC	Bug Type	Root Cause Description
BC	bc-1.06	interactive algebraic language	17K	Heap Buffer Overflow	Using wrong variable in for-loop end-condition
NCOMP	ncompress-4.2.4	file (de)compression	1.9K	Stack Smash	Fixed-length array can not hold long input file name
TAR	tar-1.13.25	GNU tar archive tool	27K	Semantic (NULL ptr)	Directory property corner case is not well handled

Table 1: Bugs tested

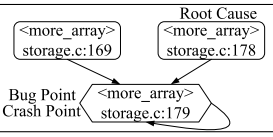
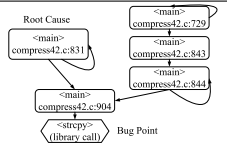
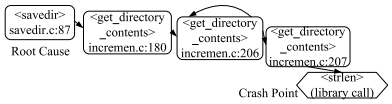
Name	Result Quality	Technique	Results from each step	Fault (Propagation) Tree
BC	Root cause in top 3	Mem-Bug Detection	Found 1 heap-overflow @0x804ee82	
	Correct fault tree	Backward Slicing	Start from 0x804ee82 Found root cause (RANK = 3)	
NCOMP	Root cause in top 6	Mem-Bug Detection	Found 1 stack-smash @0x45ba7460 (called by *comprexx*)	
	Correct fault tree	Backward Slicing	Start from 0x45ba7460–*comprexx* Found root cause (RANK = 6)	
TAR	Root cause in top 5 Correct fault tree	Mem-Bug Detection	Null pointer dereference @0x4f0f1e13 (lib. *strlen*)	
		Backward Slicing	Start from 0x4f0f1e13 Found root cause (RANK = 5)	

Table 2: Preliminary diagnosis results. (Fault trees are built in slicing step based on previous steps' result for deterministic failures.)

also not easy, because the code crosses different files and `get_directory_contents` is a large function.

3.3 Overhead

Normal-Run Since we rely on checkpoint and rollback to allow deferring complex instrumentation, there must be some overhead during normal execution. We measure two applications, Squid (network bound) and BC (CPU bound). Squid achieves 98.7% of the baseline throughput (92.5 Mbps vs 93.7 Mbps). BC achieves 99.2% of the baseline performance (53.2 seconds vs 52.8 seconds). Overhead comes from initiating the checkpoints, copy-on-write page faults, and (for Squid) copying/forwarding requests. Since the checkpoint/rollback is based on the same framework as in our previous Rx work, normal execution overhead is similar [10]. Checkpoints are in-memory using a shadow-fork() operation; since we do not need to keep many checkpoints, or keep them for long, the high overhead of writing checkpoints to disk is avoided. Given such low overhead, we can have the analysis tools standing by without hurting normal execution. In contrast, more traditional dynamic tools, such as Purify [5], impose significant overhead throughout the whole execution.

Total Time	Mem.-Bug Detection	Backward Slicing
303 sec	98 sec	205 sec

Table 4: BC failure diagnosis time

Diagnosis overhead The diagnosis tools are efficient. We present results for BC, which is a highly CPU-bound application (a worst case for our tools). Despite this, the total diagnosis time (including time to rollback) is just over 5 minutes. Among the two components, backward slicing takes the longest time, because it updates register and memory dependencies for *every* instruction. Backward slicing imposes up to 1000x overhead–completely infeasible to deploy full-time. Also, since BC is entirely CPU bound, everything it does must be traced and analyzed. For IO bound applications (i.e. servers) far fewer instructions would complete, with subsequent reductions in diagnosis time.

We should note that it is possible for diagnosis to proceed in the background, while recovery and return to normal service is performed in the foreground. Our implementation supports background diagnosis; we did not do so in order to provide isolated performance numbers for diagnosis. If ordinary activities continue in the foreground, diagnosis will take longer.

4 Conclusions and Future Work

This paper presents an innovative approach for diagnosing software failures. By leveraging lightweight checkpoint and re-execution techniques, we can capture the moment of a failure, and *automatically* perform *just-in-time* diagnosis at the end user's site. Additionally, we propose (the first to our knowledge) a *failure diagnosis protocol*, which mimics the debugging process a human programmer would take. This is well-suited to automatic on-line diagnosis *and* also helpful in in-house testing and debugging. By offloading the initial steps, it frees programmers from some of the labor of debugging.

While the TDP provides an important first step toward on-line failure diagnosis, there is, of course, more work to be done. Currently we are implementing further steps in the TDP, such as race detection, input analysis, etc.

We continue to explore the possibilities for more speculative tools. To further quantify the usefulness of the results, we intend to measure the reduction in debugging time when compared to core dumps and user descriptions. Finally, we are considering the potential of the TDP to be extended to support distributed applications.

5 Acknowledgments

We thank the anonymous reviewers for their feedback, and the Opera group members for useful discussions and proofreading. This research is supported by NSF CNS-0347854 (career award), NSF CCR-0325603, and DOE DE-FG02-05ER25688.

References

[1] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. TraceBack: First fault diagnosis by reconstruction of distributed control flow. In *PLDI*, 2005.

[2] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[3] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[4] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[5] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, 1992.

[6] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributedworm signature detection. In *USENIX Security Symposium*, 2004.

[7] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[8] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[9] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ECC-Memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[10] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies — a safe method to survive software failures. In *SOSP*, 2005.

[11] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.

[12] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, 2004.

[13] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[14] Chun Yuan, Ni Lau, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.

[15] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, 2002.

[16] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.

[17] Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Sam Midkiff, and Josep Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO*, 2004.