Cores, Debugging, and Coverage [*]

Peter Ohmann
University of Wisconsin–Madison
ohmann@cs.wisc.edu

Ben Liblit
University of Wisconsin–Madison
liblit@cs.wisc.edu

Abstract

Debugging is difficult and costly, especially for production failures. To aid developers, we enhance core memory dumps produced by crashing applications with lightweight, tunable tracing. We propose two complementary forms of tracing, path traces and global coverage, and demonstrate that they can provide substantial postmortem analysis benefit at low cost. Recent work focuses on the interplay of these mechanisms, and comparison of various forms of coverage data.

1. Problem and Motivation

Testing and debugging of software is difficult, expensive, and time-consuming. Debugging, testing, and verification can account for 50–75% of a software project's cost [12, 15, 33]. Even with extensive in-house testing, however, post-deployment failures are inevitable in complex software. In this scenario, failure reports containing traces or failure-focused views of program state are very valuable. While comprehensive reports with complete traces of production-run failures are ideal, this level of detail is impractical for complex programs. Even for simple code, full-tracing overhead may only be acceptable during in-house testing.

A core memory dump is a useful and readily-available artifact from a program crash. Coupled with symbol information, a core dump reveals the program stack for each thread at the time of termination, global variables, and some portion of heap data. Importantly, a core dump provides the crashing location and the active call-site in every other stack frame, thus providing both a snapshot of crash state and a partial call history for the program. Horwitz et al. [17] show a substantial postmortem analysis benefit by taking advantage of this information. Nevertheless, our prior work [28] indicates that significant execution ambiguity remains after postmortem analysis utilizing a stack trace alone.

The goal of this work is to enhance readily-available post-deployment failure information with lightweight, tunable instrumentation. Enhanced failure information could then be given to a developer for manual debugging, or could be used to perform automated postmortem analysis to provide a reduced view of the failure state for the developer (as in this work). Building on the information already available in a core dump can yield inexpensive but valuable postmortem data. Our existing instrumentation and analysis framework [28] validates this using two customizable core-dump-enhancement mechanisms: path traces and call-site coverage. Recent work investigates the cost and benefit of these mechanisms independently [27], and considers other forms of lightweight core dump enhancement. This document focuses on work investigating the cost/benefit trade-offs of different forms of coverage. Necessary background material is discussed in section 2. Section 3 describes our approach to lightweight instrumentation and analysis of core dump data. Section 4 discusses previous evaluations and new data comparing different granularities of coverage data. Section 5 considers related work. Section 6 concludes and suggests future and ongoing work.

2. Background

2.1 Program Slicing

Our primary postmortem analysis used for evaluation is based on program slicing. Program slicing with respect to program P, program point n, and variables V determines all other program points and branches in P which may have affected the values of V at n. In this work, we are concerned with static *closure* slices, which are a set of statements that might transitively affect V at n or the execution of n. This is contrasted with static *executable* slices [35], which are a reduction of P that, when executed on any input, preserves the values of V at n.

Ottenstein and Ottenstein [29] first proposed the program dependence graph (PDG), a useful program representation for slicing. The nodes of a PDG are the same as those in the control flow graph (CFG), and edges represent possible transfer of control or data. A *control* dependence edge represents execution flow direction via a jump or conditional branch, and a *data* dependence edge represents an assignment to a variable v at its source which may be read at its target. Thus, the backward transitive closure from n in P's PDG constitutes its *backward static slice*. Interprocedural PDG construction is more complex; our work is based on the System Dependence Graphs (SDG) of Horwitz et al. [16].

A *static* slice considers all possible program inputs and execution flows. When debugging, one would like the slice to be constrained to a particular execution. To do so, Korel and

[*] This paper originally appeared as a grand finals submission to the ACM Student Research Competition in 2014.

```
1  void foo(int x, int y, bool flag) {
2    int result = 0, result2 = 0;
3    if (y == 0)
4      y = x;
5    if (flag) {
6      result = x + y;
7      result2 = x * y;
8    } else {
9      result = x − y;
10     result2 = x / y;
11   }
12   print(result);
13   print(result2);
14 }
```

Figure 1. Slicing example. A possible dynamic slice is shaded; the static slice is framed (with or without shading).

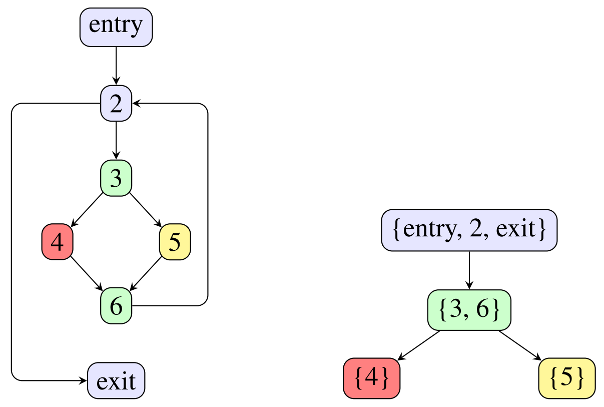
Laski [19] first proposed *dynamic* slicing; we are interested in closure (rather than executable) dynamic slices similar to those proposed by Agrawal and Horgan [3]. However, dynamic slicing can be very expensive, as it requires data equivalent to a full execution trace plus all memory accesses due to pointer variables, arrays, and structures [4, 20]. Thus, this work will lie somewhere between these two extremes: enhanced core dumps will not contain full execution traces, but could be useful in constraining a static slice over a PDG.

Figure 1 shows an example program with its static slice taken from line 13 with respect to variables {result2}. The corresponding dynamic slice for the inputs $\langle 1, 2, \mathit{false}\rangle$ is shaded. Note that it is a subset of the static slice, which summarizes all possible executions.

2.2 Basic Block Coverage and Optimization

One of our lightweight tracing mechanisms involves gathering coarse-grained coverage data. Program coverage is a common quality metric used for test suites. Testers often use statement coverage: the set of statements in the program that executed at least once. To gather statement coverage, though, one need not place instrumentation after every statement in the program; one probe per basic block is sufficient. For complete executions, coverage instrumentation can be optimized further. Agrawal [1] introduces the notion of a superblock: a set of basic blocks such that all complete function executions either execute all of the blocks or none of them. Therefore, coverage data for one basic block in each superblock is equivalent to coverage data for all basic blocks. In addition, a superblock may not require probing if its coverage can be derived from the coverage of another set of superblocks.

The algorithm to compute the superblocks for function F begins by computing the dominator and post-dominator trees for F. For two CFG nodes n and m, n dominates m if all paths from function entry to m must execute n, and n post-dominates m if all paths from m to function exit must execute n. The two trees are then merged (preserving all dominator and post-dominator edges). Each strongly-connected component of the resulting graph is collapsed into a single *superblock*; these blocks now form a DAG (the *supergraph*). To obtain 100% statement coverage, test cases need only cover the leaves of the supergraph. In order to optimize probes for gathering unknown coverage, for each superblock s with a set of child superblocks C, s must be probed if there exists a path from function entry to function exit containing s but no basic blocks in any superblock in C.



(a) CFG with annotated superblocks (b) Supergraph

Figure 2. Supergraph example

The example shown in fig. 2 has four superblocks: $\{\mathit{entry}, 2, \mathit{exit}\}$, $\{3, 6\}$, $\{4\}$, and $\{5\}$. Superblock $\{3, 6\}$, however, need not be probed, as its coverage can be derived from that of its supergraph children, $\{4\}$ and $\{5\}$.

3. Approach and Uniqueness

Recall that the high-level goal of this work is to enhance core dumps produced by crashing applications, leaving additional "breadcrumbs" to disambiguate the crashing execution corresponding to the dump. Ideally, tracing data will be useful for postmortem analysis but inexpensive to gather. Our instrumentation must be *efficient* enough (in time and space) for production use, keeping all data in memory and avoiding I/O and expensive logging. Tracing must be *customizable* after deployment (due to failure requirements or overhead sensitivity) without requiring recompilation. The data traced must *scale* with execution state, particularly readily-available core dump data (such as stack depth).

Our first tracing mechanism, titled *path tracing*, is a variant on a classic path profiling technique [8] that efficiently profiles all acyclic, intraprocedural paths. The algorithm assigns a unique integer value to each acyclic path—beginning at function entry or a loop head and ending at function exit or a back edge—and increments a global table of counters on each acyclic path execution at run time. Necessary instrumentation is optimized by finding chord edges in a maximum spanning tree, with edges weighted by path values. The final product is a table counting the number of occurrences of each path. Our variant on this approach moves all storage into the stack, giving each activation record with activated tracing a fixed-size, stack-allocated circular buffer to store the last N acyclic paths. Thus, we are able to obtain very dense information close to the point of failure in each stack frame via a stack-local execution suffix leading up to the failure point in that frame. Our focus on the failure location is a conscious choice for several reasons: this is the best place to tie to existing core dump data, empirical studies indicate that bugs tend to have short propagation distances [13, 30, 39], and developers are likely to begin from the failure point during debugging.

Path tracing data is tied directly to the program stack. This is both good and bad. The data naturally scales with stack size, and one need not maintain global or heap-allocated storage, making it efficiently maintainable. However, all data is lost immediately on a function's return. Thus, no information is maintained by this mechanism at global scope.

Coverage data provides coarser-grained global information, allowing tracing to scale gracefully as the debugging task departs from the active crash stack. Coverage can address two blind spots in path traces: execution within stack frames prior to the gathered suffix, and execution from called functions that have already returned. For the former, a fixed-size, stack-allocated array maintains one flag for each trace point, indicating whether that point was executed in the current invocation of the stack frame's function. For the latter, a global array of the same size maintains one flag for each trace point, indicating whether that point was executed in *any* invocation of each function for which tracing is active. Thus, coverage information provides crash-focused tracing in each active stack frame, and also global coverage that summarizes the data from all completed calls.

Note that the prior discussion intentionally leaves vague which program points are traced for coverage. Indeed, there are many options here with different cost/accuracy trade-offs, and the best choice is an empirical matter. Thus far, we have considered three alternatives: function coverage, call-site coverage, and statement coverage. Function coverage places one probe at the exit of each traced function. While this is the coarsest granularity (and thus the least expensive to gather), function coverage carries no intraprocedural value whatsoever; it cannot enhance the crash stack as functions in the active stack are already clearly executing. Statement coverage is gathered as basic block coverage, and can be further optimized via superblock analysis as discussed in section 2.2. This is the finest granularity considered (though "finer" granularities could include path or data-flow coverage), and thus the most expensive to gather. Call-site coverage places one probe after each relevant call instruction in traced functions. Note that it resides between the previous two extremes: if only one call in each basic block is instrumented, the probes are a subset of those for basic block coverage.

Optimization of call coverage is a reasonably straightforward extension of the algorithm from section 2.2, with some caveats. In order to optimize call-site coverage without losing data, we must specialize the set of child nodes in the supergraph when deciding whether to instrument a superblock. To be specific, for each superblock, we *can cover* the block if we *can instrument* it (i.e., it contains a call instruction) or it does not *need instrumentation*. For superblock n, we call the set of n's superblock children we can cover C'. Note that C' is a subset of C (all of n's children) from section 2.2. A block n *needs instrumentation* if there exists a path from function entry to exit containing n but containing none of C'. These definitions are mutually recursive. However, since the supergraph is always guaranteed to be a DAG [1], there exists a reverse topological ordering in which we can safely process the superblocks. The original optimization algorithm is now a special case of this refinement wherein we *can instrument* all superblocks. We *will instrument* any superblock we *can instrument* that *needs instrumentation*. For example, given the CFG shown in fig. 2, suppose that a call instruction exists in basic block 4, but none exists in 5. Then, superblock $\{3, 6\}$ must be probed, as a path now exists from entry to exit that does not contain its only *can cover* child, $\{4\}$. Coverage gathered for incomplete executions is not necessarily complete when using these optimizations. Consider the CFG shown in fig. 2, and an execution crashing in basic block 4. If optimized instrumentation is used, coverage will not be gathered for block 6, and, thus, one cannot infer whether the loop has executed prior to crashing in 4. This could decrease the utility of traced information; this is evaluated in section 4.

Tracing is customizable to meet performance requirements or to focus on specific recurring failures. We enable and disable tracing at function granularity, and replicate the body of each instrumented function. Each replica is instrumented with one set of tracing options. Simple binary modification of a global flag selects between alternatives post-deployment. In Ohmann and Liblit [28], we show that specializing tracing to particular failures significantly reduces tracing overheads.

We developed two postmortem analyses to evaluate the utility of core dump enhancement. The first, *active nodes and edges*, identifies CFG nodes and edges which could not have executed in the failing run given the crashing program stack and traced data. This analysis eliminates all unexecuted coverage points, and then performs backward control-flow reachability from the crash point, constrained by traced path data. The second analysis, *static PDG restriction*, computes a restriction of the static PDG (see section 2.1) based on the crash location and traced data. Note that a static PDG over-approximates those control and data flows actually active in any particular run. The algorithm begins by removing all unexecuted coverage points, and then maintains only dependence edges implied by the execution suffix derived from path trace data. When departing from the active stack or running out of trace, the algorithm falls back on static PDG

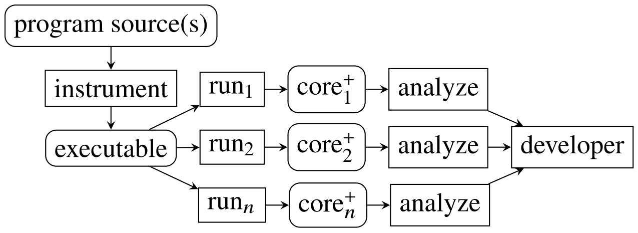


Figure 3. System organization

information. The restricted PDG can then be used to compute partially-dynamic slices from any point of interest.

Figure 3 shows an example usage scenario. The executable is instrumented once at build time, and, for each execution, tracing is customized as desired. Postmortem analysis is performed on enhanced core files, the results of which can be given to a developer to aid in debugging.

4. Results and Contributions

We implemented core dump enhancement instrumentation in a tool called csi-cc[1]. We use the LLVM compiler framework [22] to compile and instrument programs. For present evaluations, we restrict our evaluation to PDG restriction, as active nodes and edges show similar patterns. Our PDGs are produced by CodeSurfer 2.2p0 [6].

In Ohmann and Liblit [28], we show that, with realistically customized tracing, run-time overhead ranges from 0–5%, averaging only 1.3%. We traced the last 10 acyclic paths in any function appearing in any crashing stack of each application's test suite and enabled call-site coverage for all functions. To evaluate analysis effectiveness, we took interprocedural static slices from the crash point for each crashing test case, and intraprocedural static slices from the crashing location in each still-active stack frame at the time of the crash. We assessed the utility of enhanced data by comparing results against a stack-sensitive slicing approach (using the active stack to restrain possible call paths) similar to that in Horwitz et al. [17]. Postmortem analysis shows the data to be very valuable, reducing intraprocedural static slice sizes by 21–52%, and stack-sensitive interprocedural static slice sizes by 49–78% in larger applications.

In Ohmann [27], we investigated path traces and call-site coverage independently. The dense information from path traces is significantly more expensive to collect (maximum 3.5% overhead, averaging 0.8%) than call-site coverage (maximum 1.7% overhead, averaging 0.3%). Analysis results indicate that path traces tend to be preferable for intraprocedural analysis due to finer-grained information closer to the point of failure, while coverage information is preferable for interprocedural analysis where significant ambiguity exists outside the active stack. Neither, however, is completely sufficient on its own in either category, lending support to the use of multiple lightweight tracing mechanisms.

[1]`http://pages.cs.wisc.edu/~liblit/ase-2013/code/`

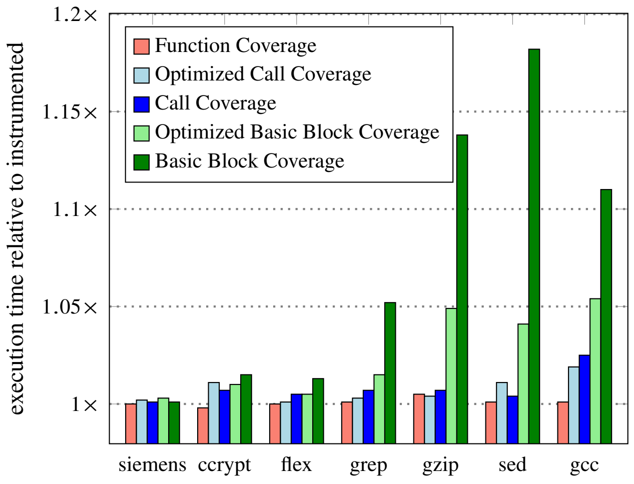
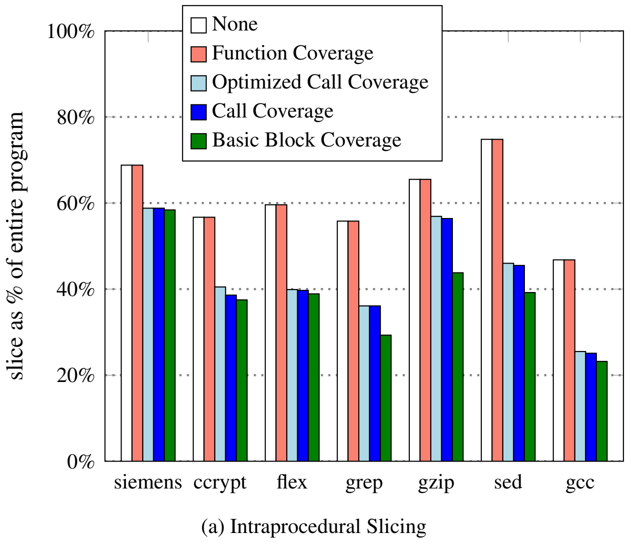


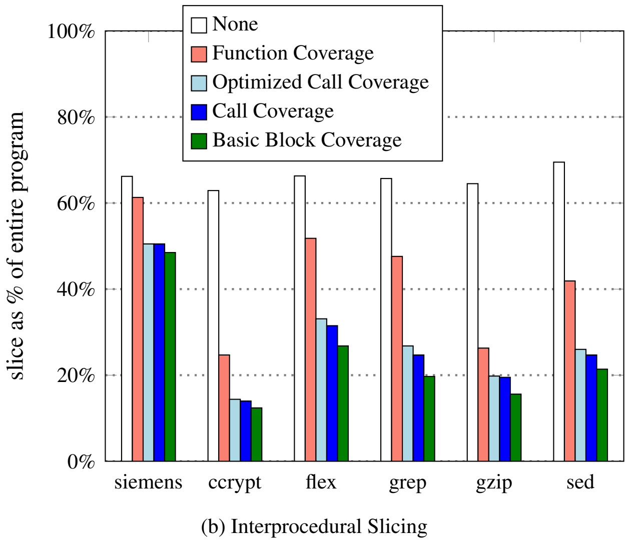
Figure 4. Run-time overhead

We recently investigated the cost/benefit trade-offs of different coverage mechanisms. For these experiments, we obtained a set of buggy applications with test suites from the Software-artifact Infrastructure Repository [32]. Some applications have multiple versions and/or faults which can be enabled independently. For the following experiments, we aggregate results across the application test suite, then by bug, by version, and finally by application so as not to over-represent particular bugs/versions causing more failures. All applications are written in C, but vary in size and functionality, ranging from 173 to 222,196 mean lines-of-code across versions. The smaller, simpler applications of the Siemens test suite are grouped for brevity in results presentation. In Ohmann [27], we showed that path tracing and coverage complement each other well. However, to avoid adding further dimensions to present evaluations, we consider only coverage alternatives independently. In addition, we do not re-evaluate the cost of customization, and instrument each function only once.

Figure 4 shows results for run-time overhead relative to uninstrumented builds (lower is better). The primary research questions are: (i) How do the coverage mechanisms compare to one another? and (ii) What benefit does coverage optimization have? For some applications (siemens, ccrypt), more detailed basic block information is not substantially more expensive than call-site coverage, at least within experimental variance. For other, larger applications (gzip, sed, gcc), though, the cost is substantial. For gzip, the cost of gathering optimized basic block coverage increases from 0.7% to 4.9% over unoptimized call-site coverage. Optimization is clearly very beneficial for basic block coverage, decreasing sed's overhead from 18.2% to 4.1%. Given that call-site coverage is significantly less expensive, the results here are less clear. Some applications (e.g., gcc) seem to indicate some benefit, but others (e.g., sed) show that variance with overheads under 1% can make results difficult to interpret.



(a) Intraprocedural Slicing



(b) Interprocedural Slicing

Figure 5. Slice reduction due to feedback analysis

Figures 5a and 5b show intraprocedural and interprocedural slicing results, respectively. These plots show slice size as a percent of the entire program (lower numbers are better), with the largest bar representing the slice size for stack-sensitive slicing with no enhanced core data. We were unable to gather interprocedural results for gcc due to the size of the whole-program PDG and our memory-bound analysis. The primary research questions are: (i) How do the coverage mechanisms compare to one another? and (ii) Does ambiguity introduced by optimization hurt results (section 3)? It is clear that, for the applications evaluated, call-site coverage can substantially reduce execution ambiguity that function coverage cannot. Thus, simply eliminating unexecuted functions outside the active stack appears to leave significant, relevant ambiguity about paths taken through those functions. Note again that function coverage cannot assist with intraprocedural analysis. The dense information from basic block coverage is clearly relevant in many cases, but the improvement here is less extreme, lending support to focus on call sites as a likely source of ambiguity (interactions between functions). With this in mind, we gathered optimized call-site coverage results. Ambiguity introduced by incomplete executions is small, though a small impact is measurable.

Overall, there are substantial trade-offs regarding coverage in our domain. While function coverage has unmeasurably small overhead, its postmortem analysis benefit is often significantly smaller than other options. Basic block coverage comes at a high overhead cost, but is useful where its cost can be tolerated. Call-site coverage provides most of the benefit of full basic block coverage at significantly less cost; thus, it is likely the best choice in many real-world deployed scenarios.

5. Related Work

Previous work uses symbolic execution with dynamic feedback data to reproduce failing executions [10, 11, 18, 31, 38]. We intentionally sacrifice perfect replay in favor of low overhead and tunable instrumentation. Symbolic execution can be very expensive and is undecidable in the general case; our analyses could provide useful constraints for a symbolic execution engine. Yuan et al. [36, 37] use static analysis with logs from failing runs to identify paths that must, may, or cannot have executed between logging points. Run-time logging provides a source of valuable information complementary to our techniques. Burger and Zeller [9] minimize failing test cases using call/return traces. Our work is similar in spirit, but focuses on lightweight tracing of deployed software.

Gupta et al. [14] compute slices within a debugger; ordered break points and call/return traces restrict the possible paths taken. While debugging is the ultimate beneficiary, our work focuses on extremely lightweight tracing with overheads low enough for production use. Nishimatsu et al. [26] shrink static slices by marking calls that execute during a given run to prune possible execution paths. Our coverage data works similarly, though our information is more detailed: we have both global coverage information as well as segregated information for each stack frame; we pair this information with more fine-grained, specialized path tracing; and we focus on customization and production failure enhancement.

Prior work optimizes coverage data by dynamically removing and customizing instrumentation [7, 34] and extending optimization to interprocedural paths [2, 21]. While not targeting in-memory core dump enhancement, these approaches provide venues for further improvement of our mechanisms.

6. Conclusions and Future Work

We have shown that substantial postmortem analysis benefit can be provided by customizable, low-cost core dump enhancement. Path tracing and global coverage provide complementary tracing strategies. We also evaluated the costs and benefits of various forms of coverage, and found that this choice can have a significant impact on run-time overhead and postmortem utility of traced data.

This work suggests a number of promising future directions. We continue to investigate new tracing mechanisms such as interprocedural call paths and dataflow hints. Reducing the cost of tracing is always a priority; thus, automatically using the static structure of programs and dynamic features of previous failures is under active investigation. We currently do not perform fault localization explicitly, but many prior efforts use slicing [5, 23, 25] and/or coverage [5, 24] for fault localization. We are interested in exploring this possibility.

7. Acknowledgments

This research was supported in part by NSF grants CCF-0953478 and CCF-1420866, and a CodeSurfer license generously provided by GrammaTech, Inc. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 25–34, New York, NY, USA, 1994. ACM.

[2] H. Agrawal. Efficient coverage testing using global dominator graphs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '99, pages 11–20, New York, NY, USA, 1999. ACM.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 60–73, New York, NY, USA, 1991. ACM.

[5] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151, Oct 1995.

[6] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Softw. Eng.*, 29(8):721–733, Aug. 2003.

[7] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE (1)*, pages 255–264. ACM, 2010.

[8] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[9] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 221–231, New York, NY, USA, July 2011. ACM.

[10] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[11] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 199–214, New York, NY, USA, 2011. ACM.

[12] B. Gauf and E. Dustin. The case for automated software testing. *Journal of Software Technology*, 10(3):29–34, Oct. 2007.

[13] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang. Characterization of Linux kernel behavior under errors. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 459–468, June 2003.

[14] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, Oct. 1997.

[15] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, Jan. 2002.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[17] S. Horwitz, B. Liblit, and M. Polishchuk. Better debugging via output tracing and callstack-sensitive slicing. *IEEE Trans. Softw. Eng.*, 36(1):7–19, Jan. 2010.

[18] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

[19] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

[20] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, Dec. 1990.

[21] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[23] Y. Lei, X. Mao, and T. Y. Chen. Backward-slice-based statistical fault localization without test oracles. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 212–221, July 2013.

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

[25] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang. Slice-based statistical fault localization. *J. Syst. Softw.*, 89:51–62, Mar. 2014.

[26] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 422–431, New York, NY, USA, 1999. ACM.

[27] P. Ohmann. CSI: Crash scene investigation. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 123–124, New York, NY, USA, 2013. ACM.

[28] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *28th International Conference on Automated Software Engineering (ASE 2013)*, Palo Alto, California, Nov. 2013. IEEE and ACM.

[29] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[30] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 235–248, New York, NY, USA, 2005. ACM.

[31] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *ICST '13: Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation*, Mar. 2013.

[32] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software–artifact infrastructure repository. `http://sir.unl.edu/portal/`, Sept. 2006.

[33] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[34] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 86–96, New York, NY, USA, 2002. ACM.

[35] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4): 352–357, July 1984.

[36] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 143–154, New York, NY, USA, 2010. ACM.

[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 3–14, New York, NY, USA, 2011. ACM.

[38] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

[39] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 251–264, New York, NY, USA, 2011. ACM.