Automated Software Engineering manuscript No.
(will be inserted by the editor)

Lightweight Control-Flow Instrumentation and Postmortem Analysis in Support of Debugging

Peter Ohmann · **Ben Liblit**

February 7, 2016

Abstract Debugging is difficult and costly. As a programmer looks for a bug, it would be helpful to see a complete trace of events leading to the point of failure. Unfortunately, full tracing is simply too slow to use after deployment, and may even be impractical during testing.

We aid post-deployment debugging by giving programmers additional information about program activity shortly before failure. We use latent information in post-failure memory dumps, augmented by low-overhead, tunable run-time tracing. Our results with a realistically-tuned tracing scheme show low enough overhead (0%–5%) to be used in production runs. We demonstrate several potential uses of this enhanced information, including a novel postmortem static slice restriction technique and a reduced view of potentially-executed code. Experimental evaluation shows our approach to be very effective. For example, our analyses shrink stack-sensitive interprocedural static slices by 53%–78% in larger applications.

Keywords postmortem program analysis, debugging, core dumps, static program slicing, path tracing, coverage

1 Introduction

Debugging is a difficult, time-consuming, and expensive part of software development and maintenance. Debugging, testing, and verification account for 50%–75% of a software project's cost (Hailpern and Santhanam 2002); these costs grow even higher in some cases (Gauf and Dustin 2007; Tassey 2002). Yet, post-deployment

P. Ohmann
University of Wisconsin–Madison
E-mail: ohmann@cs.wisc.edu

B. Liblit
University of Wisconsin–Madison
E-mail: liblit@cs.wisc.edu

failures are inevitable in complex software. When failures occur in production, detailed postmortem information is invaluable but difficult to obtain.

Developers would benefit greatly from seeing concrete traces of events leading to failures, failure-focused views of the program or program state, or suggestions of potentially-faulty statements. Sadly, full execution tracing is usually impractical for complex programs. Even for simple code, full-tracing overhead may only be acceptable during in-house testing.

One common and very useful artifact of a failed program execution is a core memory dump. Coupled with a symbol table, a core dump reveals the program stack of each execution thread at the moment of program termination, the location of the crash, the identities of all in-progress functions and program locations from which they were called, the values of local variables in these in-progress functions, and the values of global variables. Prior work with symbolic execution has shown that this information can help in deriving inputs and/or thread schedules that match a failed execution (Rößler et al 2013; Weeratunge et al 2010; Zamfir and Candea 2010).

Our goal is to support debugging using latent information in postmortem core dumps, augmented by lightweight, tunable instrumentation[1]. This paper explores four such enhancements: (1) a variant of Ball–Larus path profiling, (2) function coverage, (3) statement coverage, and (4) call-site coverage. We evaluate the trade-offs among these tracing methods, and conclude that pairing our path profiling variant with call-site coverage yields a complementary, realistic, and valuable choice for deployed applications.

Our results for this pairing with a realistically-tuned tracing scheme show low overheads (0%–5% execution time, 0%–4% dynamic memory) suitable for production use. We also demonstrate a number of potential preprocessing debugging uses of this enhanced information, including a unique hybrid program slicing restriction and a reduction of potentially-executed control-flow graph nodes and edges. These postmortem analyses can take advantage of all of our core dump enhancement tracing mechanisms. For example, one of our evaluated applications, space, crashes within a loop in a complex function containing many branches and a large **switch** statement. The bug is a missing exit statement within one **switch** case. Our analysis is able to provide the complete branch trace within the crashing function, reducing the possible set of executed statements by over 65%. This benefit comes at a tracing time overhead of just 0.3% relative to uninstrumented code. This is a simple, intraprocedural example; section 6 indicates that our approach often performs even better on larger, more complex, interprocedural cases.

This paper expands upon our previous conference paper (Ohmann and Liblit 2013) in several ways. First, we provide discussion and experimental results for varying granularities of program coverage tracing (at functions, call sites, and statements), whereas Ohmann and Liblit (2013) only discussed call-site coverage. Section 4.1.2 details this change, and provides examples. Second, and partly due to these extended tracing options, in this paper we more strongly emphasize the difference between our instrumentation (which is static) and our run-time tracing (which is customizable). We introduce the notion of a "scheme" to describe a possible tracing configuration,

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

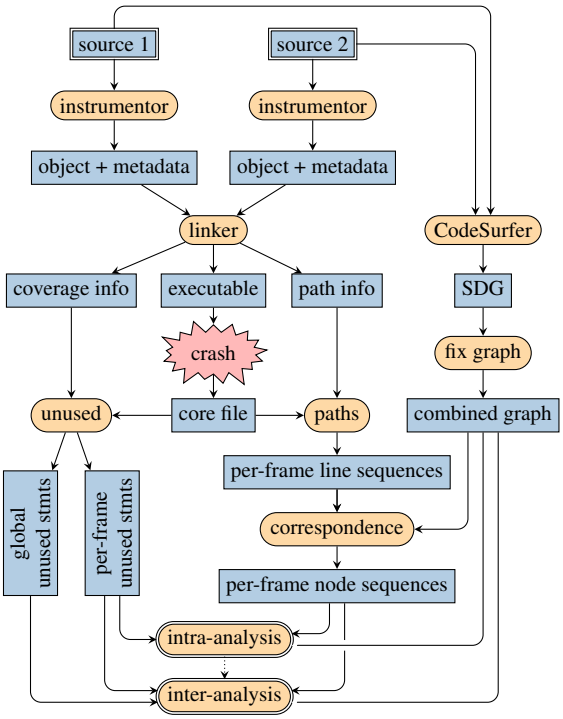


Fig. 1 Overview of data collection and analysis stages. Sharp-cornered rectangles represent inputs and outputs; rounded rectangles represent computations.

and provide a more in-depth discussion of this topic. See section 4 for details. Third, with more instrumentation choices (due to the added forms of program coverage), we provide some discussion of (and a new approach to) customization of tracing post-deployment (see section 4.2).

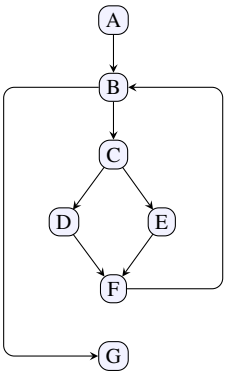
Further changes involve our analyses and evaluation. First, we provide a more detailed discussion of our control-flow node and edge reduction analysis (see section 5.1). Second, we evaluate our techniques in much greater detail. Specifically, we evaluate each tracing mechanism (including all coverage types) independently, along with tracing schemes combining multiple mechanisms, including the "realistic" scheme from Ohmann and Liblit (2013). We also assess the effect of post-deployment customization support on our tracing overhead. See sections 6.1 and 6.3 for details. Third, we provide an in-depth discussion of sources of ambiguity that we encounter in our analysis framework. Note that ambiguity in our *results* is expected: we intentionally sacrifice full-trace detail to reduce run-time overhead. However, the fact that we use two independent pieces of software for pre-instrumentation and post-crash analysis results in additional ambiguity (for matching trace data to analysis program representations). Section 6.2.2 contains full details. Finally, we set further context for our work: we discuss threats to the validity of our experiments (section 7), and provide a more extensive discussion of our future work plans (section 9).

```
void add_action(char *new_text) {
  int len = strlen(new_text);

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0)
      size += size / 8;
    else
      size = new_size;
    array = realloc_char_arr(array, size);
  }

  strcpy(&array[index], new_text);
  index += len;
}
```

(a) Code example



(b) Control-flow graph

Fig. 2 Example code

Figure 1 shows the relationships between our instrumentation and analyses; each feature of this diagram is described in the sections that follow. We begin with a motivating example in section 2, then review key background material in section 3. Section 4 describes the kinds of data we collect and our instrumentation strategies for doing so. Section 5 gives a detailed description of the analyses we perform on collected data. We assess instrumentation overhead and usefulness of analysis results in section 6. Section 7 discusses possible threats to the validity of our results. Sections 8 and 9 discuss related work and opportunities for future research. Section 10 concludes.

2 Example

Figure 2 shows an example we will refer to throughout the paper. The source code in figure 2a is taken from flex: one of the applications used for evaluation in section 6. Most often, we will make use of the function's intraprocedural control-flow graph (CFG) representation, shown in figure 2b.

3 Background

We begin by describing core dumps and their benefits for postmortem debugging. We then review a well-studied path profiling approach by Ball and Larus (1996); the present work develops a variant of this approach. Finally, we briefly outline program slicing, which serves as the basis of one of our analyses.

3.1 Core Memory Dumps

All widely-used modern operating systems can produce a "core dump" file containing a snapshot of a program's memory. A dump may be saved after abnormal program termination due to an illegal operation (such as using an invalid pointer) or on demand (such as by raising a fatal signal or failing an assertion). This can be useful if the core dump is to be used for postmortem analysis.

Typically, a core dump includes the full program stack at termination. For our purposes, the key elements are the point of failure (the exact location of the program crash), as well as the final call location in each other still-active frame on the stack (i.e., each stack frame's return address). Conveniently, core dumps are only produced in the case of program failure. Thus, collecting them imposes zero run-time overhead. This is a key advantage to using core dumps for postmortem analysis.

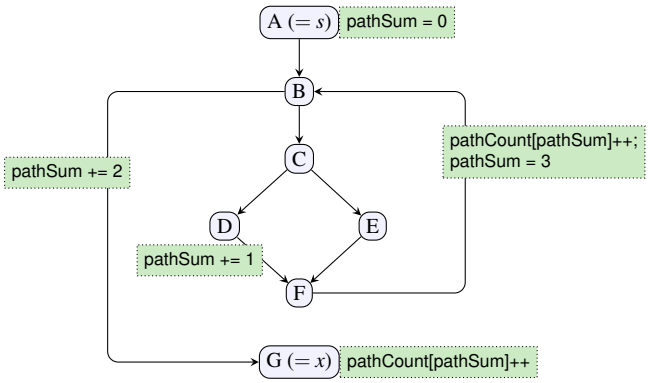
3.2 Path Profiling

Path profiling is traditionally used to compute path coverage during program testing. The approach we adopt from Ball and Larus (1996) is designed to efficiently profile all acyclic, intraprocedural paths. The algorithm first removes back edges to transform the control-flow graph (CFG) of a procedure into a directed acyclic graph (DAG). We represent the transformed CFG as a single-entry, single-exit DAG $G = (V, E, s, x)$ where V is the set of nodes in the graph and $E \subseteq V \times V$ is the set of edges with no directed cycles. Every node in V is reachable by crossing zero or more edges starting at the unique entry node $s \in V$. Conversely, the unique exit node $x \in V$ is reachable by crossing zero or more edges starting from any node. A path p through G is represented as an ordered sequence of nodes $\langle p_1, \ldots, p_{|p|} \rangle$ such that $(p_i, p_{i+1}) \in E$ for all $1 \leq i < |p|$. We define a *complete path* as a path whose initial and final nodes are s and x respectively. Let C represent the set of all complete paths; note that this set is finite since G is a DAG. Loops are handled specially, and are discussed later in this subsection.

The overall goal of the Ball–Larus algorithm is to assign a value $\mathit{Increment}(e)$ to each edge $e \in E$ such that

1. each complete path in C has a unique *path sum* produced by summing over the edges in the path;
2. the assignment is minimal, meaning that all path sums lie within the right-open interval $[0, |C|)$; and
3. the assignment is optimal, meaning that each path requires the minimal number of non-zero additions.

The first step assigns a value to each edge such that all complete path sums are unique and the assignment is minimal. To do so, the algorithm traverses the graph in reverse-topological order. For each $n \in V$ we compute $\mathit{NumPaths}[n]$, the number of paths from n to x. If we number the outgoing edges of n as $e_1, \ldots, e_k$ with respective successor nodes $v_1, \ldots, v_k$, then the weight $\mathit{Weight}(e_k)$ assigned to each outgoing edge of n is $\sum_{j=1}^{k-1} \mathit{NumPaths}[v_j]$. After this step, complete path sums using *Weight* values are unique, and the assignment is minimal.



(a) Extra instrumentation code added for path profiling

pathSum	Path
0	ABCEF
1	ABCDF
2	ABG
3	BCEF
4	BCDF
5	BG

(b) Path numbers

Fig. 3 Path profiling example

The next step optimizes the value assignment. This uses a *maximum-cost spanning tree* (MCST) of G. A MCST is an undirected graph with the same nodes as G, but with an undirected subset of G's edges forming a tree, and for which the total edge weighting is maximized. Algorithms to compute maximum-cost spanning trees are well-known. Remaining non-tree edges are *chord edges*, and all edge weights must be "pushed" to these edges. The unique cycle of spanning tree edges containing a chord edge determines its *Increment*.

Instrumentation is then straightforward. Track the path sum in a register or local variable pathSum, initialized to 0 at s. Along each chord edge e, update the path sum: pathSum += $\mathit{Increment}(e)$. When execution reaches x, increment a global counter corresponding to the path just traversed: pathCount[pathSum]++.

Cycles in the original CFG create an unbounded number of paths. Control flow across back edges requires creating extra paths from s to x by adding "dummy" edges from s to the back edge target (corresponding to initialization of the path sum when following the back edge) and from the back edge source to x (corresponding to a counter increment when taking the back edge). The algorithm then proceeds as before. Because of the dummy edges to x and from s, counter increments and reinitialization of the path sum occur on back edges. We expand our definition of a complete path to include paths that begin at back edge targets or that end at back edge sources.

Figure 3 shows possible instrumentation to profile paths in the example function from figure 2. Figure 3a shows the function's CFG annotated with pathSum and pathCount increments. Each acyclic path completes at either function exit or the loop back edge, and the counter for the path's value is incremented at that point. As shown in figure 3b, each acyclic path is uniquely numbered. Note that the assignment is clearly minimal, as each acyclic path contains at most one pathSum initialization, and one pathSum increment.

The preceding overview of path profiling focuses on details relevant to the present work; see Ball and Larus (1996) for the complete, authoritative treatment. There has been a great deal of follow-on work since the original paper (Ammons et al 1997;

Melski and Reps 1999; Sumner et al 2010; Vaswani et al 2007), some of which provides opportunities for potential future work described in section 9.

3.3 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. The original formulation by Weiser (1984) proposed the *executable static slice*: a reduction of P that, when executed on any input, preserves the values of V at n. In this work, we are concerned with non-executable or *closure slices*, which are the set of statements that might transitively affect the values of V.

Ottenstein and Ottenstein (1984) 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 CFG, and edges represent possible transfer of control or data. A *control dependence edge* is labeled either *true* or *false* and always has a control predicate or function entry as its source. An edge $n_1 \rightarrow n_2$ means that the result of the conditional at n_1 directly controls whether n_2 executes. (A node may have multiple control-dependence parents in the case of irregular control flow such as **goto**, **break**, or **continue** statements.) A *data dependence edge* is labeled with a variable v and has a variable definition at its source and a variable use at its target.

Our definition of the system dependence graph (SDG), an interprocedural dependence graph, is drawn from Horwitz et al (1988). The SDG combines all PDGs, and adds a number of new nodes and edges. Each call is now broken out into three types of nodes: a call-site, actual-in, and actual-out nodes. (We treat globals as additional parameters, following Horwitz et al (1988).) A special actual-out node is created for the return value. Each PDG is also augmented with formal-in and formal-out nodes corresponding to formal parameters and the return value, as well as global variables used or defined in the procedure. Interprocedural control dependence edges are added from each call site to the called procedure's entry node. Interprocedural data dependence edges are added for all appropriate (actual-in, formal-in) and (formal-out, actual-out) pairs, including the return value. Finally, summary edges from actual-in to actual-out nodes are computed; these represent transitive data dependence summarizing the effects of each procedure call. Details on the computation of these edges can be found in Horwitz et al (1988).

A static slice considers all possible program inputs and execution flows. While debugging, one prefers a slice that is constrained to a particular execution. Korel and Laski (1988) first proposed dynamic slicing as a solution to dataflow equations over an execution history. We are interested in closure dynamic slices similar to those proposed by Agrawal and Horgan (1990). The authors propose four variants of dynamic slicing. The first simply marks all executed nodes, and performs a static slice over that subset of the graph. The second recognizes that each executed node has exactly one control-dependence parent and one reaching definition for each variable used in the statement. Therefore, this variant slices using only dependence edges actually observed as active during the execution. The third approach recognizes that different instances of each node may have different dependence histories. Therefore, this

approach replicates each statement each time it occurs in the execution trace, attaching only the active dependence edges for that instance of the statement. Agrawal and Horgan's final approach only replicates nodes with unique transitive dependencies.

Dynamic slicing can be very expensive, potentially requiring data equivalent to a full execution trace. To make matters worse, one must trace all memory accesses due to pointer variables, arrays, and structures to have a completely accurate dynamic slice in the general case (Agrawal et al 1991; Korel and Laski 1990). Kamkar et al (1993) and Zhang and Gupta (2004) are able to reduce the cost of dynamic slicing, but the cost of fully-accurate slicing remains too high for production use. Venkatesh (1991) and Binkley et al (2006) formalize the semantics of program slicing and discuss the distinctions and orderings among the different types of program slices.

4 Data Collection

When considering which data to collect and how, several desirable properties guide our choices. Instrumentation must be **efficient** in time and space, and therefore suitable for production use. Data must be held **in memory** until failure, adding no I/O or other system calls during normal execution. Data size must **scale** with aspects of execution state, such as stack depth or number of program locations. Results must be **mappable** back to source code, and contain as little ambiguity as possible. Lastly, instrumentation must be **tunable** (for overhead or to change focus) without recompilation or redeployment.

Any core dump already records the return address of each active function at the time of failure. While this has all the above qualities, it may be insufficient on its own. Therefore, we augment core dumps with two novel techniques: path tracing and various forms of program coverage. Some of these techniques can be combined. Throughout the remainder of the paper, we will refer to a (possibly empty or singleton) set of tracing mechanisms as a tracing *scheme*.

The distinction between instrumentation and tracing is key to our technique. First, during compilation, the program must be instrumented to support each possible desired tracing scheme. Section 4.1 describes the four tracing mechanisms we consider in this paper. Second, tracing must be customizable post-deployment. We discuss the preliminary method we adopted in section 4.2. Section 4.3 touches on thread safety.

4.1 Tracing Mechanisms

This paper considers four tracing mechanisms (path tracing, statement coverage, call-site coverage, and function coverage) which we group into two different high-level methods of tracing. Our path tracing mechanism is an extension of work by Ball and Larus (1996). Coverage mechanisms are all traced similarly, while only the traced program points differ. For each instrumented function, we produce a metadata file used to interpret traced data for reconstruction and postmortem analysis; we describe this metadata individually for each tracing mechanism.

4.1.1 Ball–Larus Inspired Path Tracing

Path tracing records the last N acyclic paths taken through each function on the stack at the time of failure. Like any stack-bound data, this is discarded whenever a function returns. We achieve this using a variant of Ball–Larus path profiling. Rather than counting acyclic path executions, we instead record each completed acyclic path in a stack-allocated circular buffer.

However, completed paths alone do not yield an execution suffix. We also need the final "incomplete" path leading up to the failure. Fortunately, given a CFG G, failing node v, and a partial path sum w, we can recover the unique acyclic path that accumulates the value w upon reaching v. This is a natural consequence of the Ball–Larus approach: v and w are the only state maintained while determining acyclic paths, and therefore must constitute the system's entire "memory" of the partial path covered so far.

Formally, for any node $v \in V$, every $(\mathsf{pathSum}, v)$ pair either encodes a unique subpath in G or is infeasible. Conversely, every unique subpath in G is represented by a unique $(\mathsf{pathSum}, v)$ pair. The proof that these uniqueness properties hold is straightforward by contradiction. G has a unique exit node, x, that is reachable from v via some sequence of edges $\mathbf{E}$. This sequence of edges need not be unique. Each of those edges has been assigned an increment, and, therefore, we can compute the sum of the "suffix" sequence of edges to be $w = \sum_{e \in \mathbf{E}} \mathit{Increment}(e)$. Suppose two distinct subpaths p and q both begin at s, end at v, and share the same value of $\mathsf{pathSum}$. We can "complete the path" for both of them by connecting each subpath to $\mathbf{E}$ and getting a total path sum of $w_{\mathit{final}} = w + \mathsf{pathSum}$. However, we know that two acyclic paths do not share the same path sum by the proof from Ball and Larus (1996). It is trivially the case that no subpath can give rise to more than one possible pair as edge increments are fixed. We must merely guarantee that an accurate partial path sum is available at every point during execution, since failure can occur at any time.

Figure 4 shows appropriate instrumentation for the example from section 2. Note that the pathSum increments and pathTrace stores correspond to the path profiling instrumentation scheme shown in figure 3. Our implementation of path tracing includes a number of changes relative to standard Ball–Larus path profiling. We move array allocation into the stack, giving one trace (pathTrace) per active call. The size of this array determines how many acyclic paths are retained. This is fixed at build time, defaulting to 10. (We performed preliminary experiments on small applications, varying the buffer size over several orders of magnitude up to 100,000. We find that overhead initially increases anywhere from 10%–40% per order of magnitude. Overhead eventually stabilizes once the array is so large that most of it is unused and therefore never mapped into memory.) Note that, since space for path traces is stack-allocated, it naturally scales directly with the stack depth. Its allocation is also "free" as no explicit allocation is required, and (depending on the choice of trace size) it has minimal impact on the size of a stack frame.

The stack-allocated array serves as a circular buffer. A local variable (pathIndex) tracks the current buffer position. At each back edge and function exit, we append the path sum (pathSum) for the just-completed path to this buffer. On back edges, the path sum is reinitialized (pathSum = 3) to uniquely identify paths beginning

```
void add_action (char *new_text) {
  volatile int pathSum = 0;
  volatile int pathTrace[N];
  volatile int pathIndex = 0;

  int len = strlen(new_text);

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0) {
      size += size / 8;
      pathSum += 1;
    } else
      size = new_size;
    array = realloc_char_arr(array, size);

    pathTrace[pathIndex] = pathSum;
    pathIndex = (pathIndex + 1) % N;
    pathSum = 3;
  }

  pathSum += 2;
  pathTrace[pathIndex] = pathSum;

  strcpy(&array[index], new_text);
  index += len;
}
```

Fig. 4 Path tracing instrumentation example. Highlighted code implements path tracing.

at the loop head. Obviously, we cannot instrument functions with more paths than can be counted in a machine integer. This rarely affects 64-bit platforms, though section 6.3 notes one exception seen in our experimental evaluation. Instrumentation skips affected functions, for which we simply collect no trace data.

We must be able to access the current path sum at any point, not just at the very ends of complete paths. For safety, we forbid the compiler from keeping this value in a register. Rather, both the path sum and the trace array are declared **volatile**.

Instrumentation produces a metadata file necessary for future analyses. For each function, we record (1) a full representation of the control-flow graph with edges labeled with path sum increments; and (2) a mapping from basic blocks to line numbers. The linker aggregates this metadata into a single record for the entire executable: path info in figure 1.

4.1.2 Program Coverage

Path traces provide very detailed information close to the point of failure in each active stack frame. However, path traces have two major blind spots: old paths that have already rotated out of the circular trace buffer, and interprocedural paths through calls that have already returned.

Program coverage data can easily provide coarser-grained global information, allowing tracing to scale gracefully as the debugging task departs from the active crash

stack. Coverage instrumentation uses one global array per instrumented function, and one local array (of the same size) for each stack frame. For a function f, we select a set of statements, which we call *trace points*. These trace points are numbered $0, 1, \ldots, n-1$; these serve as indices into f's local and global coverage arrays. Coverage that we gather is *binarized*, meaning that we record whether each trace point was ever executed (1) locally, in each particular invocation of f corresponding to a stack frame; and (2) globally, for any invocation of f across the entire program's execution. Thus, each trace point corresponds to one local coverage bit per active f stack frame plus one global coverage bit. Taken together, the local and global coverage bits have several desirable properties. The local bits offer up-to-date information for trace points in each still-active function. Space for this is stack-allocated, so, like path traces, it naturally scales with the stack depth. Conversely, the global coverage bits summarize data from completed calls which have already left the stack.

Our prior work (Ohmann and Liblit 2013) considered only one set of trace points: call sites. In this work, we place that choice in better context, by also considering two alternatives. First, one may elect to gather full statement coverage. Naïvely, one trace point could be used for each statement in f. However, one trace point per basic block in f is sufficient. Second, if one is interested in function coverage, one need only select one trace point per function. Any statement guaranteed to execute on any execution of f will do; we use function entry, as selecting function exit may require either multiple trace points or adding a shared exit block. Note that function coverage is unique in that it has no stack-local variant: all functions currently in the active stack are clearly executing.

Call-site coverage is the final coverage form we consider. This mechanism is taken directly from our prior work (Ohmann and Liblit 2013). Here, we have one trace point for each call site in f. Our use of call sites as the program points for which to gather coverage information is somewhat arbitrary. However, the choice is well-matched to its purpose. Call sites mark departures from the visible call stack; these are places where stack-based tracing (such as path tracing) cannot help us. Intuitively, coverage at call sites complements dense stack-local mechanisms where that help is most likely to be useful. We find that call-site coverage works extremely well in practice (see section 6.3).

Figure 5 shows appropriate instrumentation for the example from section 2. The three variants correspond to our three sets of trace points. This example also shows some of the subsumption relationships that hold among the three types of program coverage. Call-site coverage is more precise than function coverage for this particular function: it is able to determine whether the loop was ever taken via tracing the call to realloc_char_array. The subsumption relationship, however, does not hold in general, as a function may be a leaf function (i.e., contain no calls) or not be guaranteed to execute a call instruction on every path through the function. However, statement coverage always subsumes both function and call-site coverage. In the examples of figure 5, only statement coverage distinguishes the direction of the **if** statement within the loop.

Local coverage data is stored in a stack-allocated n-element array (cov), zero-initialized at function entry. A per-function global n-element array (add_actionCov), initialized at program start, holds global coverage information. Immediately follow-

```
volatile bool add_actionCov = false;

void add_action(char *new_text) {
  add_actionCov = true;

  int len = strlen(new_text);

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0)
      size += size / 8;
    else
      size = new_size;
    array = realloc_char_arr(array, size);
  }

  strcpy(&array[index], new_text);
  index += len;
}
```

(a) Function coverage

```
volatile bool add_actionCov[3] = {false, false, false};

void add_action(char *new_text) {
  volatile bool cov[3] = {false, false, false};

  int len = strlen(new_text);
  cov[0] = add_actionCov[0] = true;

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0)
      size += size / 8;
    else
      size = new_size;
    array = realloc_char_arr(array, size);
    cov[1] = add_actionCov[1] = true;
  }

  strcpy(&array[index], new_text);
  cov[2] = add_actionCov[2] = true;
  index += len;
}
```

(b) Call-site coverage

```
volatile bool add_actionCov[6] = {false, false, false, false, false, false};

void add_action(char *new_text) {
  volatile bool cov[6] = {false, false, false, false, false, false};

  int len = strlen(new_text);
  cov[0] = add_actionCov[0] = true;

  while (len + index >= size - 10) {
    int new_size = size * 2;
    cov[1] = add_actionCov[1] = true;
    if (new_size <= 0) {
      size += size / 8;
      cov[2] = add_actionCov[2] = true;
    } else {
      size = new_size;
      cov[3] = add_actionCov[3] = true;
    }
    array = realloc_char_arr(array, size);
    cov[4] = add_actionCov[4] = true;
  }

  strcpy(&array[index], new_text);
  index += len;
  cov[5] = add_actionCov[5] = true;
}
```

(c) Statement coverage

Fig. 5 Program coverage instrumentation example. Highlighted code implements coverage.

ing each trace point i, we store **true** into slot i of both the local and global coverage arrays. To preserve ordering, the arrays are declared **volatile**.

For each trace point, we record a small amount of static metadata used to identify the trace point during analysis. In practice, our setup requires that this data differ slightly depending on the type of trace point used. Function coverage need only record the name (or mangled name) of the function. Call-site coverage records: (1) the name of the called function, if known; and (2) the line number of the call site. Statement coverage records the sequence of line numbers occurring in the basic block, and, for reasons discussed further in section 6.2.2, any calls that occur within the basic block. The linker aggregates this metadata into a single record for the entire executable: coverage info in figure 1.

4.2 Tracing Customization

In production code, it can be difficult to specify instrumentation overhead requirements beforehand, as these requirements may change over time, or vary for each program instance. Furthermore, while focusing on failure-related code could substantially reduce tracing cost, it is impossible to predict where or when post-deployment failures will occur before release. Therefore, if the cost of full program tracing is too high for production use, customizable tracing is necessary.

Our approach statically replicates each function, instruments each replica with one possible tracing scheme, and dynamically decides which replica to execute. Our original implementation from Ohmann and Liblit (2013) used *internal replication*. That is, we replicated each function body inside the function, and added a branch at function entry to select between tracing schemes. However, that work considered only two alternatives: call-site coverage with and without path tracing.

In this extended work, we allow substantially more freedom in tracing schemes, and add two new coverage alternatives. Note that the number of possible tracing schemes grows exponentially with the number of possible tracing mechanisms. Of course, some schemes are unnecessary; for example, statement coverage subsumes both call-site coverage and function coverage. Nevertheless, the number of possible schemes can quickly become unwieldy: there are 10 possibilities for our mechanisms proposed in section 4.1[2].

This explosion prompted us to instead use *external replication*; that is, we replicate each function, f, into multiple functions, one for each of f's possible tracing schemes. The original body of f is changed to a "springboard" that calls the correct variant. This significantly reduces the sizes of individual functions, and makes selected tracing schemes easier to identify (as each now constitutes its own function). However, many alternatives and optimizations are possible here. For example, one could use a **switch** statement or a jump table, inline functions or use tail calls, use a binding method to change later indirect calls to direct calls, etc. Our current approach is very straightforward, using standard **switch** statements and function calls. We rely

[2] Each of no coverage, function coverage, call-site coverage, statement coverage, and (for some functions) function coverage + call-site coverage; possibly paired with path tracing.

```
enum {
  INST_NONE,
  INST_CC,
  INST_CC_PT
} add_actionInst;

void add_action(char *new_text) {
  switch (add_actionInst) {
    default:
      add_action_NONE(new_text);
      return;
    case INST_CC:
      add_action_CC(new_text);
      return;
    case INST_CC_PT:
      add_action_CC_PT(new_text);
      return;
  }
}
```

Fig. 6 Tracing customization example

on compiler optimizations to make appropriate choices regarding inlining or conversion to jump tables. For each function, a global variable encodes which function variant to use on that particular run. These variables are stored in a special section of the data segment where they can easily be changed by direct editing of the program binary. Applications can initially ship with all instrumentation turned off. Over time, instrumentation can be activated for selected functions based on previously-observed failures.

Figure 6 shows an example for the function from section 2. The new global variable, add_actionInst, determines which tracing to use for this particular run. Here there are three possibilities: no tracing, call-site coverage tracing, or call-site coverage plus path tracing. Note that our simple approach introduces an extra indirection into each function call. This example is reasonably small; however, as mentioned previously, the number of possible tracing schemes can grow rapidly in more extreme cases. In section 6, we investigate both the memory and run-time costs of this customization.

4.3 Additional Consideration: Thread Safety

Our experimental evaluation uses only single-threaded applications, but our instrumentation remains valid with threads. Path tracing only accesses stack-allocated variables, and each thread independently maintains its own path traces. Program coverage writes to globals, but never reads from globals. (We store each coverage bit as a full byte for atomicity.) Thus, even updates to the global coverage arrays have no malign race conditions.

5 Analyses

Here we describe two analyses we developed to demonstrate the utility of the new information embedded in core dumps. First, we describe a simple algorithm that restricts the feasible execution set of control-flow graph nodes and edges based on dynamic information from a failing run. Second, we describe a novel static program dependence graph restriction algorithm which can be used without knowledge of slicing criteria to allow future restricted static program slicing. Both analyses are defined with respect to data collected as per section 4. We assume that this data has been extracted from the core file and is named and organized as follows:

***path*:** One execution suffix for each frame on the stack at program termination. Each suffix contains at least one entry: either the final crash location (for the innermost frame on the stack) or the location of the still-in-progress call to the next inner frame (for all other frames).

***localCoverage*:** One array for each (coverage mechanism $\times$ stack frame) at program termination. Array elements are Booleans, with one element per static trace point in the frame's function. If a particular form of coverage is not used, all its elements are *true*. From this we extract *unusedPoints*, the set of unexecuted trace points in each frame. Note that this description is equally valid for either call-site coverage or statement coverage. Specifically, if we only trace call-site coverage for some frame, the $localCoverage_{STMT}$ array contains all *true* elements.

***globalCoverage*:** One Boolean array for each (coverage mechanism $\times$ function) in the program, regardless of the state of the stack, with one element per static trace point in the corresponding function. If the corresponding coverage is traced, each element denotes whether or not the corresponding trace point was ever executed. Otherwise, all elements are *true*. From this we extract *globalUnusedPoints*, the set of unexecuted trace points across the entire run. Again, this description applies to all three forms of program coverage (function coverage has only a single entry per function). Unused tracing mechanisms result in wholly *true* entries.

5.1 Restriction of Execution Paths

Our first analysis determines the set of CFG nodes and edges which could not have executed given the crashing program stack and tracing data collected. This analysis involves only computing static control-flow graph reachability based on the path and coverage data. As the analysis is very light-weight, it could be used before debugging to filter portions of the program structure shown to a programmer.

Let P be a program with control-flow graph G. While the statements and edges in G represent all possible control flows on any execution of P, they are a static over-approximation of those active in any possible run of P. A full execution trace for a specific run r can precisely yield the set of executed statements and edges in G. With this information, one might reasonably restrict G to a subgraph G_r containing only the CFG nodes and edges active during r, and use the restricted subgraph during debugging or subsequent r-specific analyses.

Procedure intra_active_nodes(G_f, *path, unusedPoints*)
  input: a single-function combined graph G_f
  input: a vector of nodes $path = \langle path_1, \ldots, path_{|path|} \rangle$ representing a path in G_f
  input: a set *unusedPoints* of unexecuted trace points in G_f
  output: a set of nodes *retain*
  coverage_reduce(G_f, *unusedPoints*, $path_{|path|}$);
  retain = *path*.nodes ∪ cfg_backward_reachable(G_f, $path_1$);

Fig. 7 Intraprocedural active node analysis

Procedure coverage_reduce(G_f, *unusedPoints, last*)
  input: a single-function combined graph G_f
  input: a set *unusedPoints* of unexecuted trace points in G_f
  input: a node *last* representing the last executed node in f
  G_f.nodes −= *unusedPoints*;
  G_f.nodes = cfg_forward_reachable(G_f, f.entry) ∩ cfg_backward_reachable(G_f, *last*);

Fig. 8 Coverage reduction

If the complete execution trace is unavailable, but possible execution flows can be safely over-approximated, then the graph G_r can likewise be approximated, giving a subgraph that is larger than ideal, but still smaller than G. In our case, we have path traces and program coverage data as described in section 4. This trace data is incomplete and ambiguous: many runs can produce the same data. Our goal is to use this trace data to determine the set of possibly-*active nodes* and the set of possibly-*active edges* on *any* run that is consistent with the trace data.

Figure 7 shows the algorithm for intraprocedural active nodes analysis. We first run the procedure coverage_reduce() shown in figure 8. This procedure eliminates all trace points in the function that were not executed in a particular activation record, as well as any other program points which could not have executed given that the trace points did not execute. The procedure has two phases. First, it determines the set of nodes forward-reachable from function entry; then it finds the set of nodes backward-reachable from the function's end (in this case, the crash point). Any node not in the intersection of these two sets either (1) only executes if an eliminated trace point executes or (2) only occurs after the crash point. Then, continuing with figure 7, all nodes in the path trace must be kept, along with any nodes backward-reachable from the first path entry ($path_1$). All other nodes can be eliminated. Though not shown, determination of active edges is identical; the only difference is that we track edges crossed rather than nodes visited for each stage.

The interprocedural algorithm in figure 9 is largely an extension of the intraprocedural algorithm, with some complexities to deal with stack data. We apply the logic from figure 8 to every procedure in the entire application, now using *globalUnusedPoints*. After this, for each frame on the stack, we execute the intraprocedural algorithm over a mutable copy of the procedure's CFG, G'. This is necessary because the analysis will remove nodes from G' via a call to coverage_reduce(), and the result must respect the *retain* sets of all invocations of each procedure on the stack (in the case of recursion) and all possible invocations through transitive calls. To incorporate

```
input: a whole-program combined graph G
input: a vector of frames stack, each composed of: a vector of nodes path = ⟨path_1, ..., path_|path|⟩
       representing a path in G; and a set unusedPoints of unexecuted trace points in G
input: a mapping globalUnusedPoints from functions to a set of their unexecuted trace points
output: a set of nodes retain

forall (f, unusedPoints) in globalUnusedPoints do
    G_f = fragment of G representing function f;
    coverage_reduce(G_f, unusedPoints, f.exit);

retain = ∅;

foreach frame in ⟨stack_|stack|, ..., stack_1⟩ do
    G' = temporary copy of G restricted to frame.function;
    retain' = intra_active_nodes(G', frame.path, frame.unusedPoints);
    retain ∪= retain';

    end_call = call node located at frame.path_|frame.path|;
    calls = { n ∈ retain' | n is a call };
    if end_call ∉ cfg_backward_reachable(G', path_1) ∪ { path_1, ..., path_|path|−1 } then
        calls −= end_call;

    foreach call in calls do
        retain ∪= edges interprocedurally forward-reachable from
                  call.target without crossing any return edges;
```

Fig. 9 Interprocedural active node analysis

possible execution flows outside the visible stack, we collect the set of possibly-executed calls (excluding the final call, *end_call*, corresponding to the crash location for the relevant stack frame), and determine the set of CFG nodes that may have executed during those calls. This set is determined as all forward-reachable CFG nodes from the entry of each called function. This reachability analysis crosses call edges (to get full interprocedural information) but not return edges (to preserve context-sensitivity). Instead, we assume the intraprocedural CFG contains an intraprocedural edge corresponding to the call and return for each call site. As with the intraprocedural variant, gathering active edge information is nearly identical. Here, an additional requirement is that we also maintain a set of possible return edges (which, for this simple analysis, can be derived directly from the set of possible call edges). After all frames have completed, we can eliminate nodes and/or edges which were eliminated for all frames.

5.2 Static Slice Restriction

Our second analysis is a novel technique for program dependence graph (PDG) restriction based on an early dynamic program slicing algorithm originally proposed by Agrawal and Horgan (1990). Note, however, that we are not actually computing a dynamic slice: during analysis, the slicing criteria (program point and variables of interest) may not yet be known. Rather, we restrict the static PDG to respect the failing execution data. This can be a preparatory step for multiple future slice queries for any given slicing criteria.

input: a single-function combined graph G_f
input: a vector of nodes $path = \langle path_1, \ldots, path_{|path|} \rangle$ representing a path in G_f
input: a set *unusedPoints* of unexecuted trace points in G_f
output: a restricted version of G_f with respect to *path* and *unusedPoints*

coverage_reduce(G_f, *unusedPoints*, $path_{|path|}$);
retain = intra_control_retain(G_f, *path*) $\cup$ intra_data_retain(G_f, *path*, $\emptyset$);
G_f.pdg_edges $\cap$= *retain*;

Fig. 10 Intraprocedural dependence graph reduction

Let P be a program with dependence graph G. As with the CFG in section 5.1, dependence edges in G are a static over-approximation of those active in any possible run of P. Suppose in this case that one knew exactly which control and data dependence edges were actually used during a specific run r. One might reasonably restrict G to a dependence subgraph G_r containing only the dependence edges active during r, and use the restricted subgraph during subsequent r-specific analyses. For example, a backward static slice over G_r would yield an r-restricted dynamic slice for any program point of interest. This corresponds to approach 2 in Agrawal and Horgan (1990).

As in the CFG case, our path traces and program coverage data from section 4 allow us to over-approximate the exact set of dependence edges active in r, yielding a safe over-approximation of the ideal G_r. Specifically, we wish to compute a *trace-restricted dependence graph* that retains every dependence edge that could possibly have been active in *any* run that is consistent with the trace data.

For this formulation, we assume that G is also overlaid with the control-flow edges in each procedure (as the PDG contains all nodes from the CFG by our definition). In the remainder of the paper we refer to a graph with both CFG and PDG edges as a *combined graph*. In figures 11 to 13, "$\rightarrow$" always refers to a control-dependence (not control-flow) edge, while "$\rightarrow_v$" refers to a data-dependence edge defining v. For the high-level descriptions of the algorithms given here, we collapse all actual-in and actual-out nodes into their associated call nodes for ease of presentation.

5.2.1 Intraprocedural Restriction

Figure 10 shows the overall process of computing intraprocedural PDG restrictions, which proceeds in several phases. This algorithm resembles that in figure 7 for active CFG nodes, but determining active dependence edges is somewhat more complex. To begin, coverage information is used to prune the reachable nodes in the combined graph per figure 8, described earlier. Next, we identify the control and data dependence edges that must be retained. This process is more complex than simple reachability required for CFG nodes and edges; details for each of control-dependence and data-dependence edges appear in figures 11 and 12 respectively. Lastly, we remove all dependence edges not selected for retention.

Figure 11 shows the process for determining the retained set of control dependence edges. The goal is to identify the immediate control-dependence parent of each node in *path* and each node potentially executed prior to *path*. The vector *unattributed*

```
Function intra_control_retain(G_f, path)
    input: a single-function combined graph G_f
    input: a vector of nodes path = ⟨path_1, ..., path_|path|⟩ representing a path in G_f
    output: a set of nodes retain

    unattributed = path;
    retain = ∅;
    foreach (n, i) in (path_|path|, |path|), ..., (path_1, 1) do
        foreach p in path_{i-1}, ..., path_1 do
            if p → n is a control dependence edge in G_f then
                retain ∪= {p → n};
                remove slot i from unattributed;
                break;
    reachable = cfg_backward_reachable(G_f, path_1);
    retain ∪= {_ → n | n ∈ reachable};
    retain ∪= {q → n | q ∈ reachable ∧ n ∈ unattributed};
```

Fig. 11 Intraprocedural control-dependence retention

holds path entries for which the algorithm has yet to determine the most direct controlling node. The outer **foreach** loop walks backward (beginning from the crash point) through the entries in *path*. The inner loop begins with the entry immediately prior to the current node, again walking backward through *path*. During this inner-loop search, if a node is encountered that controls the execution of the outer-loop node, then the control dependence edge between those nodes was "active" in the traced execution, and thus must be retained. Once such a node is found, the outer-loop node has found its directly-controlling conditional; it is removed from *unattributed* and the search for that node ends. After attributing control dependence parents to as many path entries as possible, the algorithm determines the set of nodes backward-reachable from the first entry in the trace. These nodes have no additional dynamic information: any control dependence edge from a reachable node could have been active in some run producing this trace. Finally, all remaining unattributed nodes from *path* must retain all incoming control dependence edges from reachable nodes.

Determining the retained set of data dependence edges, detailed in figure 12, follows a similar process, albeit with some additions. Here, each node must determine active data dependence parents for each variable used at that node. The algorithm first determines which variables must be defined and may be used by each node in the combined graph. For brevity in presentation, *mustDef* and *mayUse* are computed as sets of (node, variable) pairs, but will also be interpreted as mappings from nodes to sets of variables. Each entry of the *unattributed* vector again corresponds to a node from the path trace, but instead tracks all unattributed variable uses at that entry. The *calleeExclusions* parameter is unused by the intraprocedural analysis. The nested loops step backward through *path*, as in control dependence retention. In this case, the outer loop finishes with a path entry only once it has attributed each variable used (or potentially used, in the case of pointers) at that node. Otherwise, at each inner loop step, data dependence edges are retained for any variables not yet attributed. Summary data dependence edges (from the appropriate actual-in to actual-out nodes) should be added to *retain* whenever a call node is encountered. The path trace does

```
Function intra_data_retain(G_f, path, calleeExclusions)
    input: a single-function combined graph G_f
    input: a vector of nodes path = ⟨path_1, ..., path_|path|⟩ representing a path in G_f
    input: a set of variables calleeExclusions unused at call site path_|path|
    output: a set of nodes retain
    mustDef = { (n, v) | n ∈ G_f.nodes ∧ n must define v };
    mayUse = { (n, v) | n ∈ G_f.nodes ∧ n may use v };
    unattributed = ⟨mayUse[path_i] for i in 1, ..., |path|⟩;
    unattributed_|path| −= calleeExclusions;
    retain = ∅;
    foreach (n, i) in (path_|path|, |path|), ..., (path_1, 1) do
        foreach p in path_{i−1}, ..., path_1 do
            if unattributed_i = ∅ then break;
            if p →_v n is a data dependence edge in G_f for some v ∈ unattributed_i then
                retain ∪= { p →_v n };
                if v ∈ mustDef[p] then
                    unattributed_i −= { v };
    reachable = cfg_backward_reachable(G_f, path_1);
    retain ∪= { _ →_v n | n ∈ reachable };
    forall (n, i) in (path_1, 1), ..., (path_|path|, |path|) do
        retain ∪= { q →_v n | q ∈ reachable ∧ v ∈ unattributed_i };
```

Fig. 12 Intraprocedural data-dependence retention

not contain data-flow information. Thus, in the case of pointers with multiple possible variable targets, the analysis cannot be certain which dependence for v was active. Therefore, the algorithm considers a used variable v attributed only if the source must always define v. Lastly, we conservatively add all possible data-dependence edges to unattributed variable uses, much as figure 11 did for control-dependence edges leading to unattributed nodes.

5.2.2 Interprocedural Restriction

Figure 13 gives the steps for interprocedural restriction. The formulation closely mirrors the interprocedural slicing method given in Horwitz et al (1988), which is also later used to slice over the restricted dependence graph. First, we use global *unusedPoints* information to remove unexecuted trace points from each function, as well as any other nodes execution-dependent on those program points.

Next we process each stack frame, beginning with the crashing function. This phase identifies active dependence edges within and between stack procedures; transitive dependencies from called (and returned) procedures are captured with summary edges. For each frame, we create G', a temporary subgraph of G containing only nodes from the frame's function. As with the active nodes analysis from figure 9, interprocedural restriction must respect the *retain* sets of all possible invocations of each procedure. We then remove unused trace points. At this point, we need to connect this frame to the previous frame by retaining data dependence edges from formal-in nodes to actual variables from the call. For the innermost frame, this has

input: a whole-program combined graph G
input: a vector of frames *stack*, each composed of: a vector of nodes $path = \langle path_1, \ldots, path_{|path|} \rangle$ representing a path in G; and a set *unusedPoints* of unexecuted trace points in G
input: a mapping *globalUnusedPoints* from functions to a set of their unexecuted trace points
output: a restricted version of G with respect to *stack* and *globalUnusedPoints*

forall (f, *unusedPoints*) in *globalUnusedPoints* **do**
  G_f = fragment of G representing function f;
  coverage_reduce(G_f, *unusedPoints*, f.exit);

$retain = \emptyset$;
$formals = \emptyset$;

foreach *frame* in $\langle stack_{|stack|}, \ldots, stack_1 \rangle$ **do**
  G' = temporary copy of G restricted to *frame*.function;
  call = call node located at *frame*.path$_{|frame.\text{path}|}$;
  coverage_reduce(G', *frame*.unusedPoints, *call*);
  actuals = variables for actual arguments for *call*;
  $connected = \{ call \rightarrow_v f \mid v \in actuals \wedge f \in formals \}$;
  $unconnected = \{ v \in actuals \mid \nexists\, call \rightarrow_v \_ \in connected \}$;
  $retain \cup= connected$;
  $retain' =$ intra_control_retain(G', *frame*.path)
    $\cup$ intra_data_retain(G', *frame*.path, *unconnected*);
  $retain \cup= retain'$;
  $formals = \{ formal \mid formal \rightarrow \_ \in retain' \}$;

worklist = all call nodes n such that *retain* contains any intraprocedural dependence edge from n;
$retain \cup=$ edges interprocedurally backward-reachable from *worklist*
    without crossing any edges from calls to formal-ins;
G.pdg_edges $\cap=$ *retain*;

Fig. 13 Interprocedural dependence graph reduction

no effect. For other frames, *connected* will contain those edges to formal-in nodes that correspond to (transitively) potentially-used formals in the previous stack frame; these must be retained. *unconnected* contains any actuals not connected to a useful formal. Note that here the intraprocedural restriction algorithms are used as subroutines. We now use the third parameter to intra_data_retain: the algorithm does not consider unused actuals to be "unattributed," as incoming data dependence edges for these variables were unused.

The final step of the algorithm retains dependence edges from transitive calls beginning from the stack frames. A *worklist* is populated with all calls not corresponding to the crash point in this frame. All dependence edges backward-reachable in the SDG from the *worklist* nodes (including edges corresponding to function returns but excluding those corresponding to function calls) must be retained. These edges correspond to transitive interprocedural dependencies for previously-returned calls. The algorithm does not need to "re-ascend" to calling procedures because summary edges are included in both phases.

5.2.3 Additional Considerations and Relationship to Dynamic Slicing

Slices over a restricted graph, like those of Agrawal and Horgan (1990) and Horwitz et al (1988), are *closure slices*. These over-approximate the set of statements that may

have affected the variable values at the chosen slice point, but are not necessarily executable or equivalent to the original program.

Unlike Agrawal and Horgan, our dependence graph restriction algorithms are not actually computing dynamic slices: they are not "slicing from" any particular program point. In fact, one way to define the analyses is as partial-trace dynamic slicing from every point along our execution suffix. The choice of static-slice start node is orthogonal to this restriction. Every static slice taken over the restricted graph should be consistent with the trace data, modulo the loss of accuracy (as in Agrawal and Horgan's approach 3) when a node is executed multiple times with different incoming dependence edges. Our dependence graph is static, so these dynamically-distinct nodes are necessarily collapsed into one static node.

It would be possible to unroll all traced paths into the dependence graph and track individual dependencies. This approach for a full execution trace produces what is known as a dynamic dependence graph, and is equivalent to Agrawal and Horgan's approach 3; our approach would produce a partial-dynamic dependence graph. While it can yield smaller dynamic slices, this approach also makes the PDG significantly larger and more complex to understand. Despite advances in compressing dynamic dependence graphs (e.g., Zhang and Gupta (2004) and the final approach by Agrawal and Horgan (1990)), graph sizes remain quite large, increasing the time and mental effort for a developer to sift through graph data to find a reasonable slice point. Thus, we do not work with dynamic dependence graphs for our analysis results; future work could consider this possibility.

Our primary goal is extremely lightweight data collection. Therefore, we do not track updates to memory locations as would be necessary for fully-accurate interprocedural dynamic slicing (Agrawal et al 1991). We accept a potential loss of accuracy that comes with static alias analysis for globals and pointer variables when crossing procedure boundaries.

6 Experimental Evaluation

We conducted experiments to assess the efficiency of our data collection strategies and the utility of the information we collect. We use Clang/LLVM 3.4 (Lattner and Adve 2004) to compile and instrument programs. Instrumentation operates directly on LLVM bitcode.

We selected a range of applications varying in functionality and size. Table 1 gives additional details about our test subjects. The Siemens applications, flex, grep, gzip, sed, and space were obtained from the Software-artifact Infrastructure Repository (Do et al 2005; Rothermel et al 2006). space contains real faults, sed contains both seeded and real faults, and the remaining SIR-provided test subjects contain only seeded faults. ccrypt and gcc are real, released versions with real faults. Some application versions have multiple faults which can be enabled separately; the "Variants" column of table 1 counts unique builds across all versions and all available faults. All of these applications are written in C. However, there are no practical reasons our approach could not be applied to object-oriented programming languages, and both our

Table 1 Evaluated applications

Application	Type	Variants	Mean LOC
print_tokens	Siemens	7	727
print_tokens2	Siemens	10	568
schedule	Siemens	9	413
schedule2	Siemens	10	373
tcas	Siemens	41	173
ccrypt	Linux utility	1	5,280
flex	Linux utility	81	14,946
grep	Linux utility	59	15,460
gzip	Linux utility	59	8,114
sed	Linux utility	75	14,314
space	ADL interpreter	38	9,563
gcc	C compiler	1	222,196

analysis back end and compiler front end support compilation and analysis of C++ code.

Results presented in this section are aggregates across all versions, bugs, and test suites of each application. In general, results vary little among builds of a given application; we note any exceptions below. We also aggregate results for all applications from the Siemens test suite to simplify presentation. These are very small, simple applications, and results indicate that they have similar results for both tracing overhead and analysis effectiveness. Again, we note exceptions below.

6.1 Overhead

Our first evaluations assess the efficiency of our tracing mechanisms and customization methodology from section 4. All experiments used a quad-core Intel Core i5-3450 CPU (3.10 GHz) with 32 GB of RAM running Red Hat Enterprise Linux 6.5.

6.1.1 Run-Time Overhead

Overhead is the ratio of execution times for instrumented and uninstrumented code. For each version of each application, we ran the test suite over the non-faulty build at least three times and took the geometric mean of the overheads for each test case. Results appear in figure 14. Smaller values are better, with 1.0 conveying no instrumentation overhead. We built each application version using our instrumentor, with all non-library functions instrumented with various instrumentation configurations.

We first evaluate each tracing mechanism individually. The first four bars (Function Coverage, Call Coverage, Statement Coverage, and Path Tracing) indicate instrumentation that does not require any customization (that is, all functions have only one variant: the particular tracing mechanism listed). Function coverage causes no measurable overhead for our test subjects. Call-site coverage is far cheaper than statement coverage (gathered as basic block coverage). The maximum overhead for call-site coverage among our test subjects was 2.0% (for gcc), while statement coverage has

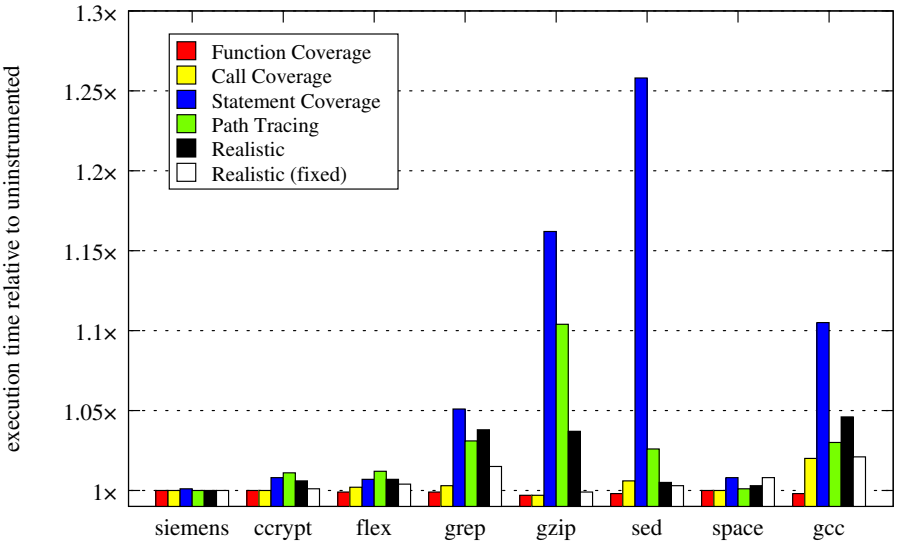


Fig. 14 Run-time overhead

overheads as high as 25.8% (for sed). gcc has thirteen functions with more than 2^{63} acyclic paths; these cannot be instrumented for path tracing. Even so, the cost of full path tracing is surprisingly low, varying across applications from negligible to 10.4% (for gzip). This suggests that our adaptation of the classic path profiling approach is very efficient, substantially reducing our overhead over full path profiling (which, according to initial results by Ball and Larus (1996), has approximately 30% average run-time overhead due to large storage requirements and the use of hashing).

Taking into account measured overheads and expected orthogonality of benefits, we then considered instrumentation based on a realistic set of tracing schemes for customization: {None, Call Coverage, Call Coverage + Path Tracing}. We then activated call-site coverage tracing for all non-library functions and path tracing for any function appearing in the crash stack of any failing test case for each application version. This is a realistic configuration if latent instrumentation can be enabled post-deployment in response to observed failures, and appears as "Realistic" in figure 14. Our results indicate that limiting path tracing to functions involved in failures can significantly reduce overhead (especially for gzip and sed). The overhead of a particular application appears to depend on non-trivial factors. For example, larger applications do not necessarily have more overhead. Most applications have comparable overheads for all versions with realistic instrumentation. One version of gzip has significantly lower overhead (about 1% on average), while the other versions are around 5%. Overheads between sed versions also vary somewhat, ranging from negligible to 2.5%. Averaged across all larger (non-Siemens) applications, the realistic configuration shows a mere 2.0% overhead.

We next evaluated what portion of the overhead for the realistic configuration was due to tracing customization (i.e., the springboard function discussed in section 4.2). The "Realistic (fixed)" bar in figure 14 shows the run-time cost of the "Realistic"

configuration had we disallowed customization (that is, had we re-instrumented each function based on observed failures to obtain the same tracing data as "Realistic" but with each function deciding at *compile time*—rather than run time—which tracing scheme to use). Clearly, customization adds a substantial portion of the run-time overhead for some applications (especially gzip). Overall, though, the "Realistic" configuration has extremely low run-time overhead ($< 5\%$ in all cases), and would be suitable for deployed applications.

All of the preceding results used non-optimized builds, as this is most conducive to debugging. We also gathered results (not shown in figure 14) using each of our previous instrumentation schemes but with Clang "-O3" optimization enabled. Analysis still works correctly on optimized code, due in part to our use of **volatile** declarations as discussed in section 4. Results for optimized code are very similar to unoptimized results, and suggest that our instrumentation does not seriously hinder program optimization. In fact, for the "Realistic" configuration with optimization, overhead averages just 1.6% across all larger applications, with a maximum overhead of 4.5% (for gzip). However, debugging optimized code is always tricky. For example, statement reordering can make the execution paths we recover difficult to understand. Prior work on debugging optimized code (Jaramillo et al 2000; Tice 1999) is directly applicable here.

6.1.2 Memory Overhead

We next measured the ratio of the maximum resident memory size of the running program for instrumented and uninstrumented code. Again, for each version of each application, we ran the test suite over the non-faulty build at least three times and took the geometric mean of the overheads for each test case. Results appear in figure 15. Smaller values are better, with 1.0 conveying no instrumentation overhead.

Again, the first four bars indicate memory overhead for each tracing mechanism individually (without customization). Our results indicate that function coverage and call-site coverage have very small memory footprints. For statement coverage and path tracing, however, extra memory usage is somewhat larger; for sed, overheads reach 8.4% for path tracing and 6.7% for statement coverage.

The next two bars ("Realistic" and "Realistic (fixed)") again correspond to the scheme proposed in section 6.1.1: optional tracing of call-site coverage and path tracing, with coverage enabled everywhere and path tracing enabled in functions appearing in failing stack traces. Beginning with the "Realistic (fixed)" scheme, it is again clear that specializing tracing to observed failures can significantly reduce overhead. In the most extreme case (sed), the memory overheads for call-site coverage and path tracing are 1.4% and 8.4% respectively (totaling 9.8%), but the uncustomized realistic configuration causes only 3.0% overhead.

However, the "Realistic" results indicate that tracing customization appears to take a large toll on static memory usage. Exploring this further, the final bar, "All Options", shows memory overhead for the pathological case where we instrument for all 10 logical tracing possibilities (as mentioned in section 4.2), but perform no tracing at execution time (i.e., we select the "none" variant for all functions). Perhaps as one might expect, the memory cost of customization is quite high. Since we create

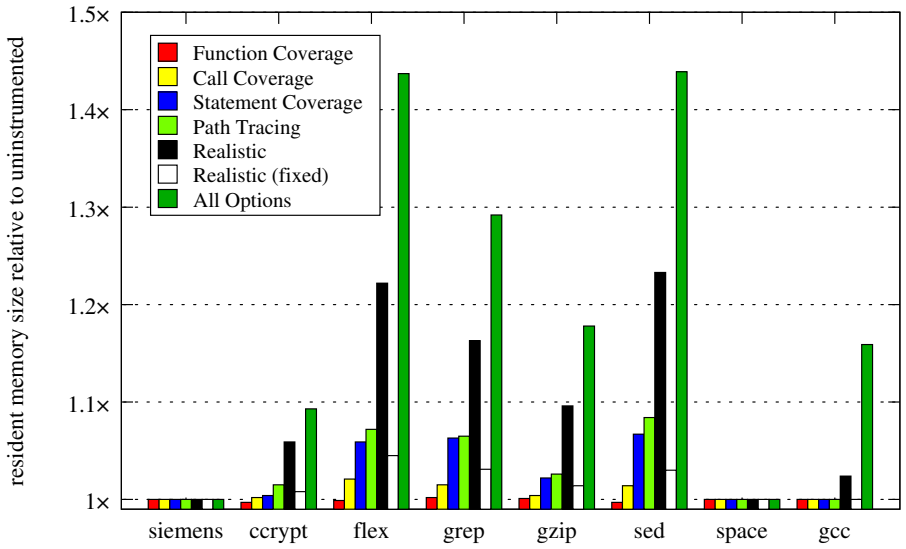


Fig. 15 Memory overhead

a new copy of every instrumented version of each function, we can potentially cause an exponential blow-up in code size. For the larger applications, the results of "All Options" instrumentation indicate that this is a potential issue. This scheme does not enable any tracing at run time, so the observed memory overheads are pure code bloat.

Nevertheless, it is important to keep this result in perspective. First, our instrumentation makes rather naïve choices; in a real-world scenario, it may be possible to make more informed decisions about which functions are likely to ever require customization or tracing in the future. Second, and most importantly, the increased memory usage for customization is a one-time cost: it does not scale throughout program execution. Thus, the *dynamic* memory cost of tracing is more closely related to the "Realistic (fixed)" results. The uncustomized realistic configuration shows just 1.8% memory overhead averaged across all larger (non-Siemens) applications.

As with time overhead, we also gathered memory overhead numbers for our tracing configurations with standard "-O3" compiler optimizations enabled. For the most part, the results are again similar to their unoptimized counterparts. Overhead numbers are slightly higher in optimized code. The differences are most pronounced for coverage mechanisms; statement coverage sees its maximum overhead value (for flex) increase from 5.9% to 12.4%. The uncustomized realistic scheme similarly sees its maximum overhead increase from 4.5% to 7.6%. Nevertheless, the average overhead for this realistic scheme with compiler optimizations (for the larger applications) is only 2.5%. Thus, our tracing has a very small memory footprint, even for optimized builds.

```
if (code != REG && !exp_equiv_p (p->exp, p->exp, 1, 0))

if (elt) elt = elt->first_same_value;
```

(a) Multiple expressions on a single line

```
if (GET_CODE (op) == CONST_INT
    && width <= HOST_BITS_PER_WIDE_INT && width > 0)

p = lookup (arg1, safe_hash (arg1, GET_MODE (arg1)) % NBUCKETS,
            GET_MODE (arg1));
```

(b) Single statements across multiple lines

Fig. 16 Examples of matching ambiguity

6.2 Analysis Implementation

This section describes details related to our implementation and evaluation of the analyses described in section 5.

6.2.1 Implementation Details

CodeSurfer 2.2p0 (Anderson et al 2003) produces our combined graphs. These match the SDG description given in section 3.3, and are overlaid with CFG edges. All CFG nodes (i.e., all nodes except for those representing "hidden" formal and actual parameters such as global variables) have associated source-code location information.

Our analysis implementation follows that given in section 5. However, there we simplified presentation by collapsing both global and local formals and actuals into their associated call node. Formals and actuals are separate nodes in CodeSurfer SDGs, and our analysis treats them separately; thus, retention can distinguish between used and unused formal and actual parameters, the *unconnected* set (figure 13) is composed of nodes (rather than variables), and summary edges exist from actual-in nodes to actual-out nodes (which are relevant for intraprocedural analysis).

6.2.2 Sources of Ambiguity

Because we use two different pieces of software (Clang and CodeSurfer) to determine statement locations for path trace entries and coverage trace points, minor disagreements are inevitable. Much of the ambiguity in matching program locations stems from the fact that line numbers are the smallest granularity at which we can reliably match Clang AST nodes to CodeSurfer graph nodes. We also find disagreements in the selection of line numbers to assign to particular program points. Naturally, our matching approach must always be conservative with respect to our analyses in order to ensure that our results safely under-approximate (but never over-approximate) the optimal reduction we can achieve.

We are generally unable to individually match different expressions occupying the same source line, as Clang and CodeSurfer may break or order the expressions differently. This means that the start, end, and size of expressions can differ, as well

as the number of nodes or operations involved. Consider the two code lines shown in figure 16a. In the first line, the LLVM bitcode contains significantly more instructions than the number of expressions given as CFG nodes produced by CodeSurfer (dereferences, the unary "!" operation, etc.). The ordering of the actual parameters in the call to exp_equiv_p and evaluation of their expressions need not correspond, so we also cannot count on an ordered many-to-one relationship.

This in-line ambiguity is particularly problematic for path traces and statement coverage data, but also has a small impact on call-site coverage matching. For path traces, we handle the ambiguity by matching each path trace entry with a set of nodes matching the given line number (rather than a single node). This set can be restricted based existing CFG edges to previous or from following entries in the trace; nevertheless, significant ambiguity is common. Specifically, we are never able to distinguish paths through a single line, such as the **if** statement given on the second line of figure 16a. Our analyses from section 5 suffer further from this problem because we often miss out on opportunities to remove a node from the *unattributed* sets (figures 11 and 12) due to not knowing if an assignment definitely executes on a particular line. Ambiguity due to the matching of LLVM line numbers to CodeSurfer nodes reduces the precision of our analysis in the (correspondence) stage of figure 1. Statement coverage, similarly, cannot always distinguish between multiple basic blocks which occur in the same line. However, as mentioned in section 4.1.2, the set of called functions is also recorded for each basic block; this can often help to distinguish blocks occurring purely within the same line. Call-site coverage is unable to distinguish between calls to the same function on the same line.

Our tools must also grapple with statements that span multiple lines. This is because Clang and CodeSurfer builds do not necessarily agree about whether to assign the line number(s) for a statement or expression to the first line, last line, or (in CodeSurfer's case) all relevant lines. Figure 16b shows two examples of statements demonstrating this issue. This issue arises most frequently with conditional expressions, ternary expressions, and calls. Actual parameters (unless they contain other expressions which will necessitate their own line number, such as another call) tend to be assigned the line number of the call statement (which is usually either the first or last line of the entire statement). Because of the great uncertainty in matching multi-line expressions, we intentionally introduce ambiguity into the combined graph to safely match Clang's output. We collapse all line numbers within each set of nodes corresponding to a multi-line expression into a single set, which we assign to all nodes of the expression. This impacts path traces, statement coverage, and call-site coverage. For path traces, this further increases the ambiguity as to which expression each path trace entry refers to on a particular line. For statement coverage, it necessitates that we only remove nodes for which all trace points corresponding to that line have "false" as their coverage bit. For call-site coverage, we similarly must ensure that call nodes are only removed if they are either the only call site on the line (for indirect calls), or the only call to the specified function (for direct calls).

Other intricate issues also necessitate some further minor introduction of ambiguity into the combined graph. For example, LLVM 3.4 assigns the line number of the close of the statement block (i.e., the closing "}" brace) to the conditional of a **do**–**while** statement. Naturally, this character has no semantic value, so it will not

appear in the CodeSurfer graph. Thus, we must include all line numbers up to the most recent statement within the loop in the set of line numbers for the loop guard conditional. These changes, as well as changes necessary for multi-line expressions, are referred to as the (fix graph) stage in figure 1. Finally, in flex, gcc, and one version of grep, we had to modify one source code line by eliminating a line break at the start of an **if** statement that otherwise caused irreconcilable disagreement between Clang and CodeSurfer line numbers.

Unfortunately, this ambiguity is quite common in the applications we examined. In fact, all four code lines from figure 16 are taken from one single function of gcc. Nevertheless, our analysis results show that we can significantly reduce ambiguity in the failing execution despite this ambiguity in our analysis framework.

6.3 Analysis Effectiveness

We evaluated the benefit of our analyses described in section 5. For test cases where core dumps were already produced, we used the generated core file. If a test case produced bad output without crashing, we used the output tracing tool of Horwitz et al (2010) to identify the first character of incorrect output, and forced the application to abort at that point. We aggregated results by taking arithmetic means across all failing tests of each faulty build, then across all faulty builds of each version. This avoids over-representing builds that simply have many failing test cases. For intraprocedural results, we ran each analysis over every function on the stack that has at least one ambiguous branch on a path from function entry to the crash point.

We ran analysis experiments by varying which tracing mechanisms were enabled. In all cases, the tracing mechanisms specified were enabled for all functions in each application. Path traces are purely intraprocedural tracing; therefore, restricting tracing to functions appearing in crashing stacks (the "realistic" configuration from section 6.1) does not result in any loss of information. As mentioned previously, gcc has thirteen functions with more than 2^{63} acyclic paths that cannot be instrumented for path tracing; however, all program coverage remains available for these functions. Due to memory constraints, we were unable to gather complete analysis results for gcc. Specifically, we excluded six gcc functions that we could not analyze with our memory-based analysis: assign_parms, expand_expr, fold, fold_truthop, rest_of_compilation, and yyparse. gcc's large size also prevented us from constructing the whole-program combined graph. Therefore, we omit interprocedural analysis results for gcc.

6.3.1 Restriction of Execution Paths

The restriction algorithms in section 5.1 can eliminate CFG nodes and edges that could not possibly have been active during a given run. Figures 17 and 18 show results (intraprocedural and interprocedural, respectively) for "active edges" as a percentage of all CFG edges. We show only results for edges here, as "active nodes" show very similar patterns. These numbers are relative to context-sensitive, stack-constrained, backward reachability. For the intraprocedural analysis, we count backward-reachable

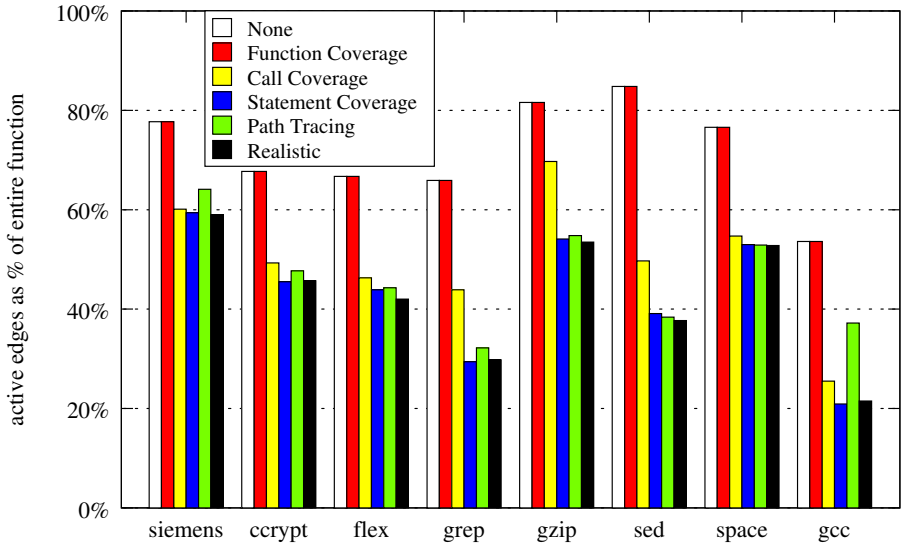


Fig. 17 Intraprocedural active edges

nodes and edges from the frame's crash point. For the interprocedural analysis, we work back from the crash point of the innermost stack frame. Smaller numbers here are better: values close to the "None" result indicate little reduction, while values closer to 0% mean that our analysis eliminated many inactive edges.

Figure 17 shows intraprocedural results. Here we measured the set of possibly-active CFG edges as a percentage of all CFG edges in each stack frame's function. We ran our analysis over every function on the stack that has at least one ambiguous branch on a path from function entry to the crash point. We first measured the reductions for each tracing mechanism individually. Reductions for the smaller Siemens applications are modest across all tracing mechanisms. Execution ambiguity is generally very low for these applications due to the small size of most functions. Results for larger applications, however, are much more impressive. Note that, by our analysis formulation from section 5.1, function coverage does not contribute to intraprocedural analysis (as all functions in the crashing stack are clearly already executing). Our other three tracing mechanisms all perform well, though complete statement coverage obtains the best results for all applications except sed (which achieves a 1% better reduction with path tracing). The high cost of full statement coverage, however, motivates consideration of the "Realistic" scheme from section 6.1: the combination of path traces and call-site coverage. Results indicate that the two mechanisms are indeed complementary. For example, gcc sees an additional 20% reduction due to the combination. The realistic configuration is the optimal choice for all but two of the applications (ccrypt and gcc), and averages 41% reduction, with a maximum reduction of 54% (sed), across the larger applications. It achieves these reductions at significantly less tracing cost than full statement coverage.

Figure 18 shows interprocedural results for active edges. Here, the plot shows the set of possibly-active CFG edges as a percentage of all edges in the entire program

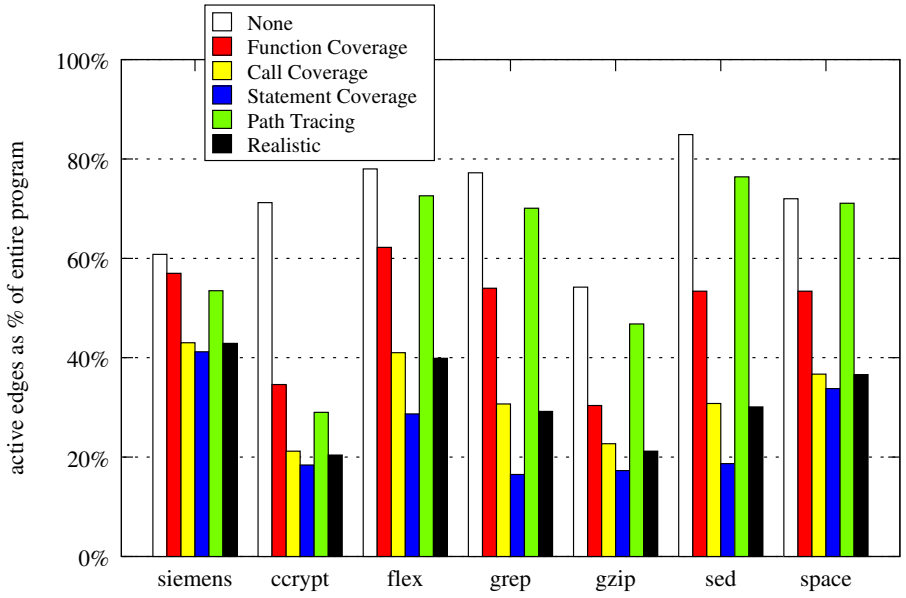


Fig. 18 Interprocedural active edges

(excluding external libraries). Again, reductions for smaller applications are modest. There are, however, some exceptions: one version of print_tokens sees an average 46% interprocedural reduction in active edges. However, in general, as with intraprocedural analysis, execution ambiguity is very low, often with only one stack frame besides main. Considering the larger applications, however, results are again much more impressive. Some patterns are clear. Coverage data is the dominating factor for interprocedural analysis. This is not surprising: coverage maintains global-scope information not available to path tracing. However, path traces do still contribute to the reduction for our "Realistic" result in all larger applications except space (which generally has very little ambiguity *within* the failing stack). Comparing our coverage mechanisms, it is clear that the coarse-grained global information provided by function coverage often still leaves a great deal of execution ambiguity that can be rectified by the finer-grained coverage mechanisms. Full statement coverage provides a clear benefit for some applications (e.g., flex, grep, and sed), but, for others, the reductions obtained for the inexpensive "Realistic" configuration are comparable.

Overall, results for the combination of path tracing and call-site coverage are quite impressive, with average reductions as high as 71% (ccrypt, interprocedural). Most applications are uniform across versions, but versions of sed have active edge reductions ranging from 38%–66% in the intraprocedural case, and 51%–85% in the interprocedural case. space versions vary from 9%–56% intraprocedurally and 6%–54% interprocedurally. In general, for complex applications, we find that a stack trace alone leaves great ambiguity as to which code was active. Our feedback data and analyses can significantly reduce this ambiguity with negligible impact on performance.

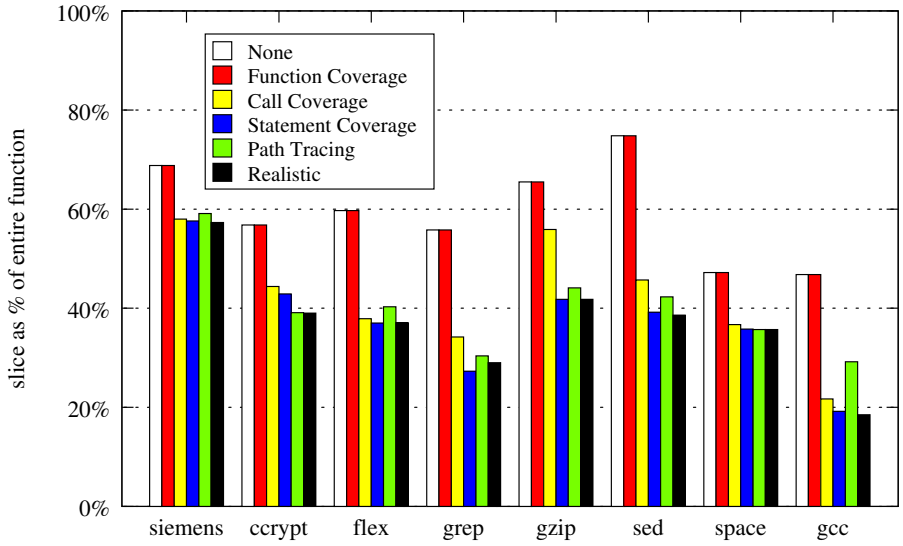


Fig. 19 Intraprocedural slicing

6.3.2 Static Slice Reduction

Our PDG restriction algorithms from section 5.2 can compute a restriction of the static PDG based on traced data. Per section 5.2.3, the computed restriction is independent of (and can be computed prior to selecting) the slicing criteria. For our evaluation, we compute interprocedural static slices backward from the crash point in the innermost stack frame; intraprocedural slices work backward from the crash point in each function in the crash stack. All interprocedural slices are callstack-sensitive (Binkley et al 2007; Horwitz et al 2010; Krinke 2004). Results show, to a large extent, very similar patterns to those for active edges.

Intraprocedural slicing results are shown in figure 19, where bars indicate the slice size for each stack frame's function as a percentage of all PDG nodes that have a source-code representation (i.e., that map to a line number). Note that a line can have more than one node. For example, for a call with multiple parameters we count each actual parameter separately, as some may be included in the slice while others are not. The "None" bar represents the slice size for a backward static slice from the crashing location in each active stack frame without the benefit of our dependence graph restriction. Smaller numbers are again better: values close to "None" indicate little reduction in slice size, while values closer to 0% mean that slices were much smaller with our restriction analysis than without. Smaller applications again see less benefit. However, larger applications again show much better results. Considering each tracing mechanism individually, full statement coverage again has the strongest results. However, the combination of path tracing and call-site coverage again performs extremely well, with the best results for all applications except grep (for which it lags behind full statement coverage by a mere 1.5% reduction). Intraprocedural

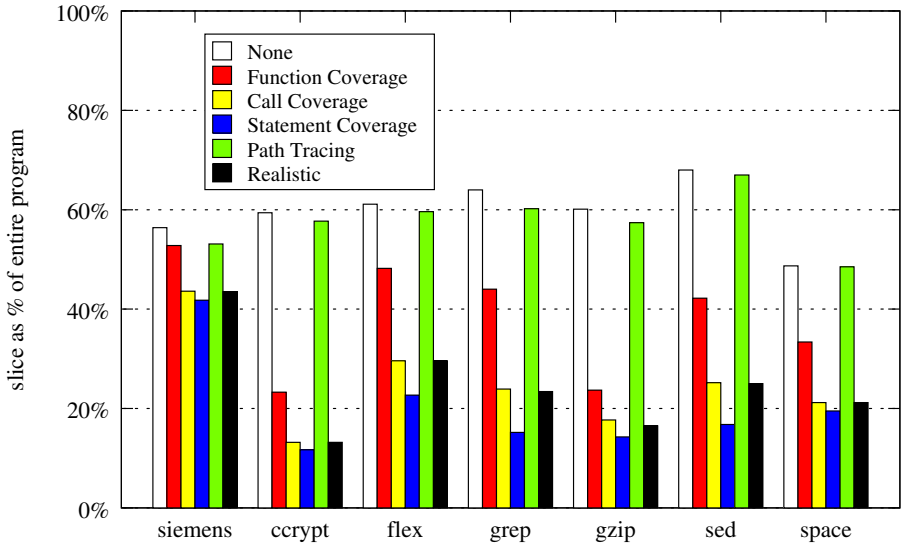


Fig. 20 Interprocedural slicing

slice reductions average 40% across all larger applications, with a maximum reduction of 53% for gcc, the largest application in our experiments.

Figure 20 shows interprocedural slicing results. Here, bars indicate the slice size as a percentage of all SDG nodes in the entire program (excluding external libraries). The "None" bar represents the slice size for a callstack-sensitive backward slice from the crashing location without the benefit of our dependence graph restriction.

As in all previous cases, the Siemens applications see only a small benefit, though there are some exceptions: one version of schedule has an average interprocedural slice reduction of 73%, but the absolute slice sizes in this particular case are small, so the absolute ambiguity is not large. Results improve substantially for larger applications, with interprocedural slice reduction showing better results (53%–78% reduction, "Realistic" trace data) than the intraprocedural variant. Coverage data is again the dominating factor in interprocedural analysis. Here, however, the benefit of full statement coverage over the combination of call-site coverage and path traces is much less pronounced. Even for grep (the application with the largest discrepancy), full statement coverage further reduces slice size by only 14% beyond the realistic tracing scheme. Overall, the realistic scheme obtains the majority of the benefit of full statement coverage at a much lower cost. space is the only larger application with highly varied results, ranging from 6%–46% intraprocedurally and 9%–62% interprocedurally.

Overall, the results for path traces and call-site coverage are again very impressive, especially interprocedurally. Even for flex, the worst among the large applications, our approach cuts interprocedural slice sizes in half. The best results, for ccrypt, show a 78% reduction, the cost of which is a mere half percent of execution time overhead ("Realistic" in figure 14).

6.4 Discussion

Our results indicate that enhancing core dumps from failing applications with lightweight, tunable tracing can yield significant postmortem analysis benefits. The combination of path traces and call-site coverage proved an inexpensive, complementary, and effective pairing to enhance postmortem analyses. Path traces have the additional benefit of providing a detailed (though incomplete) partial trace leading up to the point of failure. This would likely be very valuable to a developer in a real-world scenario, but we could not assess this benefit in our present experimental setting. Future work could consider performing a live debugging study with real developers to gauge the complete benefit of path traces and the reductions from our coverage mechanisms.

There are clearly 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. Full statement 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 statement coverage at significantly less cost; thus, it is likely the best choice in many real-world deployed scenarios.

Overall, our goal was to show that a large benefit can be drawn from very little cost via targeted core dump enhancement; we have succeeded in this regard. A particular real-world application may benefit (in both overhead cost and analysis performance) from a more customized tracing scheme than the rather simple schemes we consider here. While our "Realistic" scheme proved widely applicable, different scenarios will allow different choices. If higher execution overheads can be tolerated, statement coverage proved helpful for many applications. If execution constraints are tightened, simple function coverage can still yield significant benefit in many scenarios. To this end, our approach is intentionally customizable.

7 Threats to Validity

We attempted to gather fair and generalizable results, but have not formally proven the correctness of our approaches or implementation. Here we discuss threats to the validity of our results, and measures taken to mitigate these risks.

7.1 Threats to Internal Validity

We began with a claim that nearly all deployed software will have bugs throughout its lifetime. Our software is no exception. Bugs in our algorithm design or software implementation could impact the correctness of analysis results or the accuracy of overhead results.

We made significant attempts to control other factors (outside of our tracing mechanisms) which could have impacted overhead results. We used a single machine and ran our tests under minimal load. Nevertheless, our instrumentation does impact code size, stack frame size, and (in rare cases where we need to place instrumentation along edges) the control-flow graph of the programs. Thus, it is possible that other factors not directly related to instrumentation may impact our results.

Section 6.2.2 notes changes we required in order to match static information between the two tools we used for our analysis experiments. The intentional addition of this ambiguity into our combined graphs makes our results a safe over-approximation of the optimal result, but could impact the relationship between the results of different tracing methods. In particular, it may have a larger effect on some tracing methods when compared with others. From a subjective and cursory inspection, this seems to impact our path traces most significantly.

7.2 Threats to External Validity

While we attempted to select applications with a wide variety of size and functionality, it is obviously impossible for us to test our approach on all possible programs. Thus, our results may not generalize to all deployed software. In particular, we evaluate only applications written in the C programming language in this work, so the applicability of our approaches to modern object-oriented languages cannot be guaranteed.

Many applications had seeded faults, raising typical concerns as to whether such faults are realistic. All applications include test suites with both failing and successful runs; we distinguished these by comparing the result with that produced by a non-buggy reference version of the same application. In real deployed applications, it may be more difficult to identify failures or to obtain failing core dumps (either due to not recognizing failures until later in execution or due to security concerns). While this does not directly impact the utility of our traced information, it could make it more difficult for a developer to select appropriate functions for instrumentation/tracing, or impact the "distance" between the failure and the fault. Longer fault propagation distances likely *increase* the benefit of traced information, but the overall impact is not clear from our lab experimental setting.

8 Related Work

Several prior efforts use symbolic execution in conjunction with dynamic feedback data to reproduce failing executions (Cao et al 2014; Crameri et al 2011; Jin and Orso 2012; Rößler et al 2013; Zamfir and Candea 2010). We intentionally sacrifice perfect replay in favor of low overhead and tunable instrumentation. As symbolic execution can be very expensive and is undecidable in the general case, we see related work on symbolic execution based on core dumps as possible beneficiaries of the restriction analyses we perform. Yuan et al (2010, 2011) use static analysis with logs from failing runs to identify paths that must, may, or cannot have executed between logging points. Clause and Orso (2007) track environment interactions for replay and minimization of failing executions. While we do not require run-time logging or tracing of environment interactions, these approaches may provide additional valuable sources of information that could be used in conjunction with the analyses described here. Failure reconstruction is one possible postmortem analysis task worthy of study; we propose other inexpensive pre-debugging analyses in this work, and demonstrate

their effectiveness in reducing failing execution ambiguity. Manevich et al (2004) use backward dataflow analysis to reproduce failing executions based on only a failure location and typestate information regarding the failure. While very efficient, this approach is geared toward solving specific typestate problems with very simple types (e.g., tracking NULL values for null-pointer dereferences). Our approach uses denser information, but targets a wider range of unknown failures: anything that can be made to dump core.

Adaptive bug isolation (Arumuga Nainar and Liblit 2010) and the Gamma project (Bowring et al 2002; Orso et al 2002) emphasize adaptive post-deployment instrumentation with data collection aggregated across large user communities. Such approaches are complementary to our own: we focus on gathering very valuable information at very low cost, while these related efforts focus on how best to deploy information-gathering instances.

Gupta et al (1997) compute slices within a debugger; ordered break points and call/return traces restrict the possible paths taken. While Gupta et al focus on interactive debugging, our approach is intended for deployed applications. This imposes different requirements, leading to different solutions. Our overheads must remain small relative to a completely uninstrumented application, not merely relative to an application running in an interactive debugger. Gupta et al use complete break-point and call/return traces, while we have only bounded buffers for each morsel of dynamic data. Takada et al (2002) offer near-dynamic slicing by tracking each variable's most recent writer. Our work focuses more on control than data; in the presence of pointers and arrays, lightweight dynamic data dependence tracing in the style of Takada et al could be a useful addition. Call-mark slicing (Nishimatsu et al 1999) marks calls that execute during a given run, then uses this to prune possible execution paths, thereby shrinking static slices. The first phase of our interprocedural slice restriction algorithm uses a similar strategy. However, our information is more detailed: we have both global coverage information as well as segregated information for each stack frame.

9 Future Work

Our results indicate that the core dump enhancement approach has great potential to aid in postmortem debugging. In this section, we consider some of the most promising future directions for continued research.

9.1 Unused Information

The astute reader will note that, while our analyses can significantly reduce execution ambiguity, we cannot claim to make full use of the information we gather. First, we only make use of *false* coverage bits to eliminate unused code. It is clear that *true* coverage bits can also provide important execution information; for example, **if** statement branches not contained within a loop will always have only one branch covered for local execution. Second, our present analysis does not make use of function coverage bits for intraprocedural analysis, but it is possible to do so. All direct calls to

any unexecuted function could also be removed as part of our coverage restriction procedure from figure 8. Finally, our traced information holds great promise for more heavyweight analyses. Particularly, our data could be used as additional constraints to reconstruct failing executions via symbolic execution. Recent work has shown great strides in this area (Jin and Orso 2012; Rößler et al 2013; Zamfir and Candea 2010). Future work could also consider aggregation of data from multiple failing runs in, for example, slice-based fault localization (e.g., Lei et al (2012)) or some form of union slicing (e.g., Mulhern and Liblit (2008)).

9.2 Customization

Customization currently makes up a large portion of both the run-time and memory overheads of the realistic instrumentation configuration we propose. Future work could take at least two possible routes to mitigate this. First, instrumentation could eliminate tracing options for some functions by user customization (based on a more limited set of expected future tracing needs) or by making more intelligent static decisions (if it can be determined that certain functions are better or worse candidates for specific tracing mechanisms). Second, as we note in section 4.2, we use a very simple switch-based customization method for our current implementation. Future work could consider other indirection techniques, such as function multi-versioning for Scenario-Based Optimization (Mars and Hundt 2009).

9.3 Tracing Extensions

Our results indicate that specializing path tracing to functions involved in previous failures can substantially reduce overhead. However, when path tracing is deployed more widely, our preliminary inspection indicates that a large portion of its overhead comes from functions with a very large number of acyclic paths. In addition, we were unable to make use of path tracing instrumentation for a non-trivial number of functions from one of our test subjects due to very large path counts. One might simply leave these uninstrumented; unfortunately, these complex functions may be exactly what the programmer needs help understanding. One could also trace just some paths, perhaps adapting work by Apiwattanapong and Harrold (2002) or Vaswani et al (2007) on focused path profiling variants. The resulting trace suffix would be ambiguous but potentially still useful.

Our global program coverage mechanisms work well as described here, but are both coarse-grained and inflexible. We are interested in approaches that can encode calling context with low overhead (Bond and McKinley 2007; Sumner et al 2010), rather than explicitly and blindly logging all trace points. We are also interested in leveraging aspects of data flow as well as control flow; analyses by Yuan et al (2011) to identify “most-useful” variables may be a good start. Our current instrumentation and analysis techniques should be able to analyze C++ applications; we are interested in exploring whether our techniques translate well to larger object-oriented software with many dynamically-bound calls.

10 Conclusion

Our primary design goal was to provide valuable extended core-dump information for debugging with low enough overhead to be used in a production setting. Our adaptations of path tracing and program coverage are complementary strategies that realize this goal. Experimental evaluation finds interprocedural slice reductions as high as 78%, and active node and edge reductions as high as 71%. Average run-time overheads are merely 1.2% in a realistic debugging configuration, with a maximum overhead of less than 5%. Thus, we provide significant debugging support for negligible cost.

Acknowledgements Mark Chapman provided invaluable assistance with understanding and manipulating CodeSurfer-generated PDGs.

Compliance with Ethical Standards — Funding This research was supported in part by DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0953478, CCF-1217582, and CCF-1420866; a grant from the Wisconsin Alumni Research Foundation; 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

Agrawal H, Horgan JR (1990) Dynamic program slicing. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '90, pp 246–256, DOI 10.1145/93542.93576, URL http://doi.acm.org/10.1145/93542.93576

Agrawal H, DeMillo RA, Spafford EH (1991) Dynamic slicing in the presence of unconstrained pointers. In: Proceedings of the Symposium on Testing, Analysis, and Verification, ACM, New York, NY, USA, TAV4, pp 60–73, DOI 10.1145/120807.120813, URL http://doi.acm.org/10.1145/120807.120813

Ammons G, Ball T, Larus JR (1997) Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '97, pp 85–96, DOI 10.1145/258915.258924, URL http://doi.acm.org/10.1145/258915.258924

Anderson P, Reps T, Teitelbaum T (2003) Design and implementation of a fine-grained software inspection tool. IEEE Trans Softw Eng 29(8):721–733, DOI 10.1109/TSE.2003.1223646, URL http://dx.doi.org/10.1109/TSE.2003.1223646

Apiwattanapong T, Harrold MJ (2002) Selective path profiling. In: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM, New York, NY, USA, PASTE '02, pp 35–42, DOI 10.1145/586094.586104, URL http://doi.acm.org/10.1145/586094.586104

Arumuga Nainar P, Liblit B (2010) Adaptive bug isolation. In: Kramer J, Bishop J, Devanbu PT, Uchitel S (eds) ICSE (1), ACM, pp 255–264

Ball T, Larus JR (1996) Efficient path profiling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, IEEE Computer Society, Washington, DC, USA, MICRO 29, pp 46–57, URL http://dl.acm.org/citation.cfm?id=243846.243857

Binkley D, Danicic S, Gyimóthy T, Harman M, Kiss A, Korel B (2006) A formalisation of the relationship between forms of program slicing. Sci Comput Program 62(3):228–252, DOI 10.1016/j.scico.2006.04.007, URL http://dx.doi.org/10.1016/j.scico.2006.04.007

Binkley D, Gold N, Harman M (2007) An empirical study of static program slice size. ACM Trans Softw Eng Methodol 16(2), DOI 10.1145/1217295.1217297, URL http://doi.acm.org/10.1145/1217295.1217297

Bond MD, McKinley KS (2007) Probabilistic calling context. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, ACM, New York, NY, USA, OOPSLA '07, pp 97–112, DOI 10.1145/1297027.1297035, URL http://doi.acm.org/10.1145/1297027.1297035

Bowring J, Orso A, Harrold MJ (2002) Monitoring deployed software using software tomography. In: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM, New York, NY, USA, PASTE '02, pp 2–9, DOI 10.1145/586094.586099, URL http://doi.acm.org/10.1145/586094.586099

Cao Y, Zhang H, Ding S (2014) SymCrash: selective recording for reproducing crashes. In: Crnkovic I, Chechik M, Grünbacher P (eds) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, ACM, pp 791–802, DOI 10.1145/2642937.2642993, URL http://doi.acm.org/10.1145/2642937.2642993

Clause J, Orso A (2007) A technique for enabling and supporting debugging of field failures. In: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '07, pp 261–270, DOI 10.1109/ICSE.2007.10, URL http://dx.doi.org/10.1109/ICSE.2007.10

Crameri O, Bianchini R, Zwaenepoel W (2011) Striking a new balance between program instrumentation and debugging time. In: Proceedings of the Sixth Conference on Computer Systems, ACM, New York, NY, USA, EuroSys '11, pp 199–214, DOI 10.1145/1966445.1966464, URL http://doi.acm.org/10.1145/1966445.1966464

Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal 10(4):405–435

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

Gupta R, Soffa ML, Howard J (1997) Hybrid slicing: integrating dynamic information with static analysis. ACM Trans Softw Eng Methodol 6(4):370–397, DOI 10.1145/261640.261644, URL http://doi.acm.org/10.1145/261640.261644

Hailpern B, Santhanam P (2002) Software debugging, testing, and verification. IBM Syst J 41(1):4–12, DOI 10.1147/sj.411.0004, URL http://dx.doi.org/10.1147/sj.411.0004

Horwitz S, Reps T, Binkley D (1988) Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '88, pp 35–46, DOI 10.1145/53990.53994, URL http://doi.acm.org/10.1145/53990.53994

Horwitz S, Liblit B, Polishchuk M (2010) Better debugging via output tracing and callstack-sensitive slicing. IEEE Trans Softw Eng 36(1):7–19, DOI 10.1109/TSE.2009.66, URL http://dx.doi.org/10.1109/TSE.2009.66

Jaramillo C, Gupta R, Soffa ML (2000) FULLDOC: A full reporting debugger for optimized code. In: Proceedings of the 7th International Symposium on Static Analysis, Springer-Verlag, London, UK, UK, SAS '00, pp 240–259, URL http://dl.acm.org/citation.cfm?id=647169.718156

Jin W, Orso A (2012) BugRedux: reproducing field failures for in-house debugging. In: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE 2012, pp 474–484, URL http://dl.acm.org/citation.cfm?id=2337223.2337279

Kamkar M, Fritzson P, Shahmehri N (1993) Three approaches to interprocedural dynamic slicing. Microprocessing and Microprogramming 38(1-5):625–636

Korel B, Laski J (1988) Dynamic program slicing. Inf Process Lett 29(3):155–163, DOI 10.1016/0020-0190(88)90054-3, URL http://dx.doi.org/10.1016/0020-0190(88)90054-3

Korel B, Laski J (1990) Dynamic slicing of computer programs. J Syst Softw 13(3):187–195, DOI 10.1016/0164-1212(90)90094-3, URL http://dx.doi.org/10.1016/0164-1212(90)90094-3

Krinke J (2004) Context-sensitivity matters, but context does not. In: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop, IEEE Computer Society, Washington, DC, USA, SCAM '04, pp 29–35, DOI 10.1109/SCAM.2004.7, URL http://dx.doi.org/10.1109/SCAM.2004.7

Lattner C, Adve V (2004) 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

Lei Y, Mao X, Dai Z, Wang C (2012) Effective statistical fault localization using program slices. In: Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference, IEEE Computer Society, Washington, DC, USA, COMPSAC '12, pp 1–10, DOI 10.1109/COMPSAC.2012.9, URL http://dx.doi.org/10.1109/COMPSAC.2012.9

Manevich R, Sridharan M, Adams S, Das M, Yang Z (2004) PSE: Explaining program failures via postmortem static analysis. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, SIGSOFT '04/FSE-12, pp 63–72, DOI 10.1145/1029894.1029907, URL http://doi.acm.org/10.1145/1029894.1029907

Mars J, Hundt R (2009) Scenario based optimization: A framework for statically enabling online optimizations. In: Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009, IEEE Computer Society, pp 169–179, DOI 10.1109/CGO.2009.24, URL http://doi.ieeecomputersociety.org/10.1109/CGO.2009.24

Melski D, Reps TW (1999) Interprocedural path profiling. In: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Springer-Verlag, London, UK, UK, CC '99, pp 47–62, URL http://dl.acm.org/citation.cfm?id=647475.727620

Mulhern A, Liblit B (2008) Effective slicing: A generalization of full and relevant slicing. Tech. Rep. 1639, University of Wisconsin–Madison

Nishimatsu A, Jihira M, Kusumoto S, Inoue K (1999) Call-mark slicing: an efficient and economical way of reducing slice. In: Proceedings of the 21st International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '99, pp 422–431, DOI 10.1145/302405.302674, URL http://doi.acm.org/10.1145/302405.302674

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

Orso A, Liang D, Harrold MJ, Lipton R (2002) Gamma system: continuous evolution of software after deployment. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA '02, pp 65–69, DOI 10.1145/566172.566182, URL http://doi.acm.org/10.1145/566172.566182

Ottenstein KJ, Ottenstein LM (1984) 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, ACM, New York, NY, USA, SDE 1, pp 177–184, DOI 10.1145/800020.808263, URL http://doi.acm.org/10.1145/800020.808263

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

Rothermel G, Elbaum S, Kinneer A, Do H (2006) Software-artifact infrastructure repository. URL http://sir.unl.edu/portal/

Sumner WN, Zheng Y, Weeratunge D, Zhang X (2010) Precise calling context encoding. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, New York, NY, USA, ICSE '10, pp 525–534, DOI 10.1145/1806799.1806875, URL http://doi.acm.org/10.1145/1806799.1806875

Takada T, Ohata F, Inoue K (2002) Dependence-cache slicing: A program slicing method using lightweight dynamic information. In: Proceedings of the 10th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, IWPC '02, pp 169–, URL http://dl.acm.org/citation.cfm?id=580131.857015

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

Tice CM (1999) Non-transparent debugging of optimized code. PhD thesis, EECS Department, University of California, Berkeley, URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/6232.html

Vaswani K, Nori AV, Chilimbi TM (2007) Preferential path profiling: compactly numbering interesting paths. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '07, pp 351–362, DOI 10.1145/1190216.1190268, URL http://doi.acm.org/10.1145/1190216.1190268

Venkatesh GA (1991) The semantic approach to program slicing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '91, pp 107–119, DOI 10.1145/113445.113455, URL http://doi.acm.org/10.1145/113445.113455

Weeratunge D, Zhang X, Jagannathan S (2010) Analyzing multicore dumps to facilitate concurrency bug reproduction. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XV, pp 155–166, DOI 10.1145/1736020.1736039, URL http://doi.acm.org/10.1145/1736020.1736039

Weiser M (1984) Program slicing. IEEE Trans Softw Eng 10(4):352–357, DOI 10.1109/TSE.1984.5010248, URL http://dx.doi.org/10.1109/TSE.1984.5010248

Yuan D, Mai H, Xiong W, Tan L, Zhou Y, Pasupathy S (2010) 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, ACM, New York, NY, USA, ASPLOS XV, pp 143–154, DOI 10.1145/1736020.1736038, URL http://doi.acm.org/10.1145/1736020.1736038

Yuan D, Zheng J, Park S, Zhou Y, Savage S (2011) Improving software diagnosability via log enhancement. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XVI, pp 3–14, DOI 10.1145/1950365.1950369, URL http://doi.acm.org/10.1145/1950365.1950369

Zamfir C, Candea G (2010) Execution synthesis: a technique for automated software debugging. In: Proceedings of the 5th European Conference on Computer Systems, ACM, New York, NY, USA, EuroSys '10, pp 321–334, DOI 10.1145/1755913.1755946, URL http://doi.acm.org/10.1145/1755913.1755946

Zhang X, Gupta R (2004) Cost effective dynamic program slicing. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '04, pp 94–106, DOI 10.1145/996841.996855, URL http://doi.acm.org/10.1145/996841.996855