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

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

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

***Abstract*—Debugging is difficult and costly. As a human 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 in 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, such as shrinking stack-sensitive interprocedural static slices by 49–78% in larger applications.

I. 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 [16]; these costs grow even higher in some cases [14, 36]. Yet, post-deployment 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 matching a failed execution [32, 40, 44].

Our goal is to support debugging using latent information in postmortem core dumps, augmented by lightweight, tunable instrumentation[1]. This paper explores two such enhancements:

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

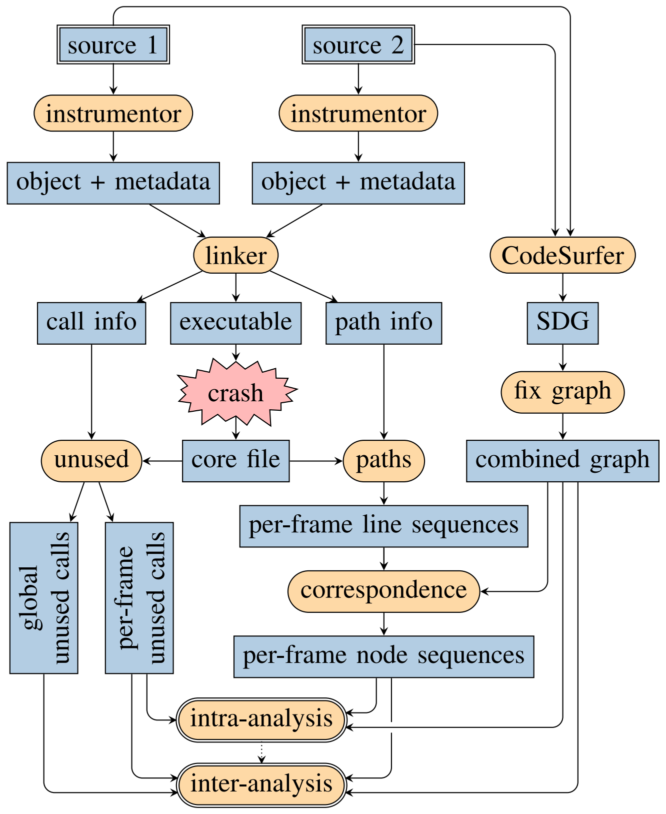

Fig. 1. Overview of data collection and analysis stages. Sharp-cornered rectangles represent inputs and outputs; rounded rectangles represent computations.

(1) a variant of Ball–Larus Path Profiling and (2) simple call-site coverage. Our results with a realistically-tuned tracing scheme show low overheads (0–5%) suitable for production use. We also demonstrate a number of potential pre-processing 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.

Figure 1 shows the relationships between our instrumentation and analyses. After reviewing background material (section II), we describe each feature of the diagram. Section III describes the kinds of data we collect and our instrumentation strategies for doing so. Section IV gives a detailed description of the analyses we perform on collected data. We assess instrumentation overhead and usefulness of analysis results in section V. Section VI discusses related work and section VII concludes.

II. Background

We begin by describing core dumps and their benefits for postmortem debugging. We then describe a well-studied path profiling approach developed by Ball and Larus [7], a variant of which we develop for this work. Finally, we briefly outline program slicing on which we base one of our analyses.

A. Core Memory Dumps

All widely-used modern operating systems can produce a file containing a "core dump" of main 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 contains 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, they impose no run-time overhead to collect: a key advantage to the use of core dumps for postmortem analysis.

B. Path Profiling

Path profiling is traditionally used to compute path coverage during program testing. The approach we adopt from Ball and Larus [7] 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 $1 \le i < |p|$. We define a *complete path* as a path whose initial and final nodes are s and x respectively. 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 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 range $(0, |p|-1)$; 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.

The next step optimizes the value assignment. This requires computing 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. The path sum is kept in a register or 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: pathCoverage[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 beginning at back edge targets or ending at back edge sources.

Our overview focuses on details relevant to the present work; see Ball and Larus [7] for the complete, authoritative treatment. There has been a great deal of follow-on and related work since the original paper [3, 27, 34, 38], some of which provides opportunities for potential future work described in section VII.

C. 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 [41] 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 [31] 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 $n_1 \to n_2$ 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. [17]. This graph 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 as in Horwitz et al. [17].) 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 globals used or defined. 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. [17].

A static slice considers all possible program inputs and execution flows. While debugging, one would like the slice to be constrained to a particular execution. Korel and Laski [22] 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 [1]. 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 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 [2, 23]. Kamkar et al. [21] and Zhang and Gupta [45] are able to reduce the cost of dynamic slicing, but the cost of fully-accurate slicing remains too high for production use. Venkatesh [39] and Binkley et al. [8] formalize the semantics of program slicing and discuss the distinctions and orderings among the different types of program slices.

III. 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 call-site coverage.

A. 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 failing CFG 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. 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 2 shows a small instrumentation example. To instrument each function, we first clone the entire function body. One copy is instrumented; the other is left unchanged. A new branch at function entry chooses between the two. For each function, a global flag (fooInstumentationActivated) encodes whether or not to use the instrumented version on that particular run. These flags 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.

Our implementation of path tracing includes a number of changes relative to standard Ball–Larus path counting. 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.)

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

```
foo() {
  while (...)
    if (...)
      call a();
    else
      call b();
}
```

(a) Original

```
volatile bool fooCalls[2] = {false, false};
bool fooInstumentationActivated;

foo() {
  if (fooInstumentationActivated) {
    volatile int pathSum = 0;
    volatile int pathTrace[N];
    volatile int pathIndex = 0;
    volatile bool calls[2] = {false, false};
    while (...) {
      if (...) {
        call a();
        calls[0] = fooCalls[0] = true;
        pathSum += 1;
      } else {
        call b();
        calls[1] = fooCalls[1] = true;
      }
      pathTrace[pathIndex] = pathSum;
      pathIndex = (pathIndex + 1) % N;
      pathSum = 3;
    }
    pathSum += 2;
    pathTrace[pathIndex] = pathSum;
  } else
    // original body of foo
}
```

(b) Instrumented

Fig. 2. Instrumentation example. Highlighted code implements path tracing and call-site coverage respectively.

identify paths beginning 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 V-B 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 fig. 1.

B. Call-Site Coverage

Call-site coverage addresses two blind spots in path traces: paths prior to the first in the trace buffer, and interprocedural paths through calls that have already returned. Nishimatsu et al. [29] gather coverage at call sites executed during a particular run and use this to restrict the static program dependence graph. Similarly, we keep one global bit for each call site indicating whether that call ever executed during a run. We also track call-site coverage for each active frame in the program stack. Taken together, the local and global coverage bits have several desirable properties. The local bits offer up-to-date information for call sites in each still-active function. Space for this is stack-allocated, so it naturally scales directly with the path trace. Conversely, the global coverage bits summarize data from completed calls which have already left the stack.

Figure 2 shows a small instrumentation example. Call-site coverage uses one global array per instrumented function, and one local array (of the same size) for each stack frame. For a function f, we number its call sites $f_0 \dots f_{n-1}$; these serve as indices into f's local and global coverage arrays. Local coverage data is stored in a stack-allocated n-element array (calls), zero-initialized at function entry. A per-function global n-element array (fooCalls), initialized at program start, holds global call-site information. Immediately following each call site f_i, we store **true** into slot i of both the local and global coverage arrays. To preserve ordering, the arrays and stores are declared **volatile**. For our experiments, we always enable call-site coverage for all functions, as our evaluations demonstrate that it is inexpensive to do so.

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 the stack-based path trace cannot help us. Thus, coverage at call sites complements path tracing where that help is most likely to be useful. We find that call-site coverage works extremely well in practice (see section V-B).

For each call site, we record two pieces of static metadata: (1) the name of the called function, if known; and (2) the line number of the call site. The linker aggregates this metadata into a single record for the entire executable: call info in fig. 1.

C. 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. Call-site coverage writes to globals, but never reads from globals. (We store each call-site coverage bit as a full byte for atomicity.) Thus, even updates to the global call-site coverage arrays have no malign race conditions.

IV. 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 by which the feasible execution set of control-flow graph nodes and edges is restricted 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 criterion to allow future restricted static program slicing. Both analyses are defined with respect to data collected as per

```
Procedure callsite_reduce($G_f$, unusedCalls, last)
    input: a single-function combined graph $G_f$
    input: a set unusedCalls of unexecuted call nodes in $G_f$
    input: a node last representing the last executed node in $f$
    $G_f$.nodes −= unusedCalls;
    $G_f$.nodes = cfg_forward_reachable($G_f$, $f$.entry)
            ∩ cfg_backward_reachable($G_f$, last);
```

Fig. 3. Call-site reduction

```
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 unusedCalls of unexecuted call nodes in $G_f$
output: a restricted version of $G_f$ with respect to path and
        unusedCalls
callsite_reduce($G_f$, unusedCalls, $path_{|path|}$);
retain = intra_control_retain($G_f$, path)
     ∪ intra_data_retain($G_f$, path, ∅);
$G_f$.pdg_edges ∩= retain;
```

Fig. 4. Intraprocedural dependence graph reduction

section III. 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. All paths contain 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).

***callCoverage*:** One array for each stack frame at program termination. Array elements are Booleans, with one element per static call site in the frame's function. If call-site coverage is not used, all elements are *true*. From this we extract *unusedCalls*, the set of unexecuted call nodes in each frame.

***globalCallCoverage*:** One Boolean array for each function in the program, regardless of the state of the stack, with one element per static call site in the corresponding function. If call-site coverage is used, each element denotes whether or not the corresponding call site was ever taken. Otherwise, all elements are *true*. From this we extract *globalUnusedCalls*, the set of unexecuted call nodes across the entire run.

A. 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 call coverage data. As the analysis is very light-weight, it could be used before debugging to eliminate portions of the program structure shown to a programmer.

In the intraprocedural case, we first run the algorithm in fig. 3. This algorithm eliminates all call sites in the function that were not taken in a particular activation record, as well as any other program points which could not have executed given that the call sites did not execute. The algorithm proceeds in 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 (a) only executes if an eliminated call site executes or (b) only occurs after the crash point. Then, 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 are eliminated. Elimination of edges is identical; the only difference is that we track edges crossed rather than nodes visited.

The interprocedural algorithm is a straightforward extension. We apply the logic from fig. 3 to every procedure in the entire application, now using *globalUnusedCalls*. After this, for each frame on the stack, we execute the intraprocedural algorithm over a mutable copy of the CFG. The only difference is that, at call sites, we explore both intraprocedural and interprocedural CFG edges. After all frames have completed, we eliminate nodes and edges which were eliminated for all frames.

B. 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 [1]. 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.

Let P be a program with dependence graph G. Dependence edges in G are a static over-approximation of those active in any possible run of P. Suppose one knew exactly which control and data dependence edges were actually used during a specific run r. Then one might reasonably restrict G to a subgraph G_r containing only the dependence edges active during r, and use the restricted subgraph during subsequent r-specific analyses. This corresponds to approach 2 in Agrawal and Horgan [1].

If the exact dependence edges are not known, but 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 call coverage data as described in section III. This trace data is incomplete and ambiguous: many runs can produce the same data. 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 overlain 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 figs. 5 to 7, "$\rightarrow$" 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.

```
Function intra_control_retain($G_f$, $path$)
    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 $G_f$
    output: a set of nodes $retain$
    $unattributed$ = $path$;
    $retain$ = $\emptyset$;
    foreach ($n$, $i$) in $(path_{|path|}, |path|), \ldots, (path_1, 1)$ do
        foreach $p$ in $path_{i-1}, \ldots, path_1$ do
            if $p \rightarrow n$ is a control dependence edge in $G_f$ then
                $retain$ $\cup$= $\{p \rightarrow n\}$;
                remove slot $i$ from $unattributed$;
                break;
    $reachable$ = cfg_backward_reachable($G_f$, $path_1$);
    $retain$ $\cup$= $\{\_ \rightarrow n \mid n \in reachable\}$;
    $retain$ $\cup$= $\{q \rightarrow n \mid q \in reachable \wedge n \in unattributed\}$;
```

Fig. 5. Intraprocedural control-dependence retention

```
Function intra_data_retain($G_f$, $path$, $calleeExclusions$)
    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 of variables $calleeExclusions$ unused at call site
           $path_{|path|}$
    output: a set of nodes $retain$
    $mustDef$ = $\{(n, v) \mid n \in G_f$.nodes $\wedge$ $n$ must define $v\}$;
    $mayUse$ = $\{(n, v) \mid n \in G_f$.nodes $\wedge$ $n$ may use $v\}$;
    $unattributed$ = $\langle mayUse[path_i]$ for $i$ in $1, \ldots, |path| \rangle$;
    $unattributed_{|path|}$ −= $calleeExclusions$;
    $retain$ = $\emptyset$;
    foreach ($n$, $i$) in $(path_{|path|}, |path|), \ldots, (path_1, 1)$ do
        foreach $p$ in $path_{i-1}, \ldots, path_1$ do
            if $unattributed_i = \emptyset$ then break;
            if $p \rightarrow_v n$ is a data dependence edge in $G_f$ for
            some $v \in unattributed_i$ then
                $retain$ $\cup$= $\{p \rightarrow_v n\}$;
                if $v \in mustDef[p]$ then
                    $unattributed_i$ −= $\{v\}$;
    $reachable$ = cfg_backward_reachable($G_f$, $path_1$);
    $retain$ $\cup$= $\{\_ \rightarrow_v n \mid n \in reachable\}$;
    forall ($n$, $i$) in $(path_1, 1), \ldots, (path_{|path|}, |path|)$ do
        $retain$ $\cup$= $\{q \rightarrow_v n \mid q \in reachable \wedge v \in unattributed_i\}$;
```

Fig. 6. Intraprocedural data-dependence retention

1) Intraprocedural Restriction: Figure 4 shows the overall process of computing intraprocedural restrictions, which proceeds in several phases. To begin, call-site information is used to prune the reachable nodes in the combined graph per fig. 3, described earlier. Next, we identify the control and data dependence edges that must be retained; details for each of these appear in figs. 5 and 6 respectively. Lastly, we remove all dependence edges not selected for retention.

Figure 5 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* 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 searching process, 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 fig. 6, follows a similar process, albeit with some additions. Here, each node must determine active data dependence parents for each variable used. 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. The *unattributed* vector now tracks all unattributed variable uses at each entry. The *calleeExclusions* parameter is unused by the intraprocedural analysis. The nested loops, as in control dependence retention, step backward through *path*. 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 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 fig. 5 did for control-dependence edges leading to unattributed nodes.

2) Interprocedural Restriction: Figure 7 gives the steps for interprocedural restriction. The formulation closely mirrors the interprocedural slicing method given in Horwitz et al. [17], which is also later used to slice over the restricted dependence graph. First, we use global *unusedCalls* information to remove unexecuted calls from each function, as well as any other nodes execution-dependent on those calls.

```
input: a whole-program combined graph $G$
input: a vector of frames $stack$, each composed of: a vector of
       nodes $path = \langle path_1, \dots, path_{|path|} \rangle$ representing a path in
       $G$; and a set $unusedCalls$ of unexecuted call nodes in $G$
input: a mapping $globalUnusedCalls$ from functions to a set of
       their unused call nodes
output: a restricted version of $G$ with respect to $stack$ and
        $globalUnusedCalls$

forall ($f$, $unusedCalls$) in $globalUnusedCalls$ do
    $G_f$ = fragment of $G$ representing function $f$;
    callsite_reduce($G_f$, $unusedCalls$, $f$.exit);

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

foreach $frame$ in $\langle stack_{|stack|}, \dots, stack_1 \rangle$ do
    $G'$ = temporary copy of $G$ restricted to $frame$.function;
    $call$ = call node located at $frame.\mathrm{path}_{|frame.\mathrm{path}|}$;
    callsite_reduce($G'$, $frame$.unusedCalls, $call$);
    $actuals$ = 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. 7. Interprocedural dependence graph reduction

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 make a temporary subgraph of G containing only nodes from the frame's function. This is done because interprocedural restriction 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. We then remove unused calls. 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 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.

3) Additional Considerations and Relationship to Dynamic Slicing: Slices over a restricted graph, like those of Agrawal and Horgan [1] and Horwitz et al. [17], 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 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. 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 dependence parents. Our dependence graph is static, so these dynamically-distinct nodes are necessarily collapsed into one static node. The choice of static-slice start node is orthogonal to this restriction.

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 [2]. We accept a potential loss of accuracy that comes with static alias analysis for globals and pointer variables when crossing procedure boundaries.

TABLE I
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

V. 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 3.1 [25] to compile and instrument programs. Instrumentation operates directly on LLVM bitcode.

We selected a range of applications varying in functionality and size. Table I gives additional details about our test subjects. The Siemens applications, flex, grep, gzip, sed, and space were

TABLE II
EXECUTION TIME RELATIVE TO UNINSTRUMENTED CODE

Application	None	All		Realistic	Optimized
		−Calls	+Calls		
print_tokens	1.000	0.999	1.002	1.001	0.998
print_tokens2	1.002	1.001	1.000	0.999	0.999
schedule	1.000	1.000	1.001	0.999	0.999
schedule2	1.001	1.000	1.000	1.000	1.000
tcas	1.000	1.000	1.001	1.001	1.000
ccrypt	1.003	1.008	1.016	1.005	0.999
flex	1.003	1.006	1.006	1.008	1.025
grep	1.019	1.049	1.053	1.032	1.020
gzip	1.024	1.155	1.157	1.044	1.011
sed	1.009	1.030	1.037	1.015	1.000
space	1.000	1.004	1.003	1.002	1.001
gcc	1.027	1.062	1.080	1.053	1.015

obtained from the Software-artifact Infrastructure Repository [33]. 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 I 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 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.

A. Run-Time Overhead

Overhead is the ratio of execution times for instrumented and uninstrumented code. We measured overheads at various levels of tracing, using a quad-core Intel Core i5 with 16 GB of RAM running Red Hat Enterprise Linux 6.3. 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 table II. 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. We then varied (1) the set of functions whose instrumentation is enabled at run time, and (2) whether or not call-site coverage was used. The "None" column of table II shows the overhead with all functions instrumented, but all tracing disabled at run time. The "All" columns show the overhead with path tracing enabled for all functions in the program, either without ("–Calls") or with ("+Calls") call-site coverage enabled. The "Realistic" column represents a compromise between "None" and "All": call-site coverage is activated, and path tracing is enabled for any function appearing in the crash stack of any failing test case for that version. This is a realistic configuration if latent instrumentation can be enabled post-deployment in response to observed failures.

All of the preceding results used non-optimized builds, as this is most conducive to debugging. The "Optimized" column of table II activates the same instrumentation as "Realistic" 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 III. Overheads drop to 2.5% for the single slowest application (flex), and a mere 0.6% averaged across all applications. 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 [19, 37] is directly applicable here.

Our results indicate that limiting path tracing to functions involved in failures can significantly reduce overhead. Call-site coverage imposes little overhead in most cases: it would be reasonable to enable this unconditionally for all functions. 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 2% on average), while the other versions are around 5%. Overheads between sed versions vary more, ranging from negligible to 4.5%. Averaged across all applications, the realistic configuration shows a mere 1.3% overhead.

B. Analysis Effectiveness

We evaluated the benefit of our analyses described in section IV. 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. [18] 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 all analysis experiments with tracing and call-site coverage enabled for all functions. gcc has thirteen functions with more than 2^{63} acyclic paths; these cannot be instrumented for path tracing, but all call-site coverage remains available. gcc's large size also prevented us from constructing the whole-program combined graph. Therefore, we omit interprocedural analysis results for gcc. We also 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.

1) Implementation Details: CodeSurfer 2.2p0 [4] produces our SDGs. All CFG nodes (i.e., all nodes except for those representing "hidden" actuals such as global variables) have associated source-code location information.

Because we use two different pieces of software (Clang and CodeSurfer) to determine statement locations for path trace entries and call sites, minor disagreements are inevitable. Line numbers are the smallest granularity at which we can reliably match Clang AST nodes to CodeSurfer graph nodes. Because of this, ambiguity reduces the precision of our analysis in the correspondence stage of fig. 1. 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. Our analysis also introduces ambiguity into the SDG to safely match Clang's output, referred to as the fix graph stage in fig. 1.

2) Restriction of Execution Paths: The restriction algorithms in section IV-A can eliminate CFG nodes and edges that could not possibly have been active during a given run. The four "Active Nodes" and "Active Edges" columns in table III report the reduction in the number of CFG-reachable nodes and edges in our experiments. These numbers are relative to context-sensitive, stack-constrained, backward reachability. For the intraprocedural analysis, we count backward-reachable 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. Larger numbers here are better: 0% means no reduction, while values closer to 100% mean that our analyses eliminated many inactive nodes or edges.

Reductions for the smaller applications are modest. Most failures in these applications occur very early in execution. Execution ambiguity is very low, often with only one stack frame besides main. There are exceptions: one version of print_tokens2 sees an average 49% reduction in active edges. Results for larger applications are much more impressive, with average reductions as high as 71%. Most applications are uniform across versions, but versions of sed have active edge reductions ranging from 38–83% in the intraprocedural case, and 51–85% in the interprocedural case. space versions vary from 9–56% intraprocedurally and 6–53% 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.

3) Static Slice Reduction: Per section IV-B3, the computed restriction is independent of (and can be computed prior to selecting) the slicing criteria. 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 [9, 18, 24].

The two "Slice" columns in table III show our results. These numbers represent reduction in slice sizes relative to a callstack-sensitive backward slice from the same location without the benefit of our dependence graph restriction. Larger numbers are better: 0% means no reduction in slice size, while values closer to 100% mean that slices were much smaller with our restriction analysis than without. Slice sizes count 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 parameters we count each actual parameter separately, as some may be included in the slice while others are not.

Smaller applications again see less benefit. As before, there are some exceptions: one version of schedule has an average interprocedural slice reduction of about 75%, but the absolute slice sizes here are small, so the absolute ambiguity is not large. space is the only larger application with highly varied results, ranging from 6–46% intraprocedurally and 10–63% interprocedurally. Results improve substantially for larger applications, with interprocedural slice reduction showing better results (49–78% reduction) than the intraprocedural variant. Call-site coverage has the potential to eliminate many functions from the combined graph; close examination shows that this happens frequently and to great effect. In addition, path tracing plays an important role: if a trace can determine that frame-local call sites were not taken, the slice is able to remain within the stack frame where ambiguity is resolved more directly. flex is a good counterexample: path traces commonly do not reach function entry (being stuck in a tight loop), and we see less impressive numbers both intra- and interprocedurally. Yet even in this case, the worst among the large programs, our approach cuts interprocedural slice sizes almost in half. The best results, for ccrypt, show nearly an 80% reduction, the cost of which is a mere half percent of overhead ("Realistic" in table II).

VI. Related Work

Several prior efforts use symbolic execution in conjunction with dynamic feedback data to reproduce failing executions [12, 13, 20, 32, 44]. 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. [42, 43] use static analysis with logs from failing runs to identify paths that must, may, or cannot have executed between logging points. While we do not require run-time logging, it provides another valuable source of information that could be used in conjunction with the analyses described here.

Selective path profiling [5], adaptive bug isolation [6], and the Gamma project [11, 30] 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. [15] 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

TABLE III
RELATIVE REDUCTION IN COUNTS OR SIZES DUE TO FEEDBACK ANALYSIS

	Intraprocedural			Interprocedural		
Application	Active Nodes %	Active Edges %	Slice %	Active Nodes %	Active Edges %	Slice %
print_tokens	23	27	23	37	39	35
print_tokens2	31	35	23	21	22	18
schedule	8	11	9	17	19	22
schedule2	5	7	0	8	8	19
tcas	22	24	22	53	57	24
ccrypt	33	36	39	71	71	78
flex	31	36	35	50	50	49
grep	45	49	51	61	61	63
gzip	38	42	43	58	59	72
sed	50	54	49	65	66	65
space	24	28	21	45	46	53
gcc	44	49	52	-	-	-

break-point and call/return traces, while we have only bounded buffers for each morsel of dynamic data. Takada et al. [35] 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 [29] 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.

VII. Conclusions and Future Work

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. Path tracing and call-site 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 overheads are merely 1.3% in a realistic debugging configuration. Thus, we provide significant debugging support for negligible cost.

We consider several areas open for improvement. Our preliminary inspection suggests that the bulk of our overhead comes from path tracing in complex functions. 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 Vaswani et al. [38] on preferential path profiling. The resulting trace suffix would be ambiguous but potentially still useful. Global call-site coverage works well as described here, but is both coarse-grained and inflexible. We are interested in approaches which can encode calling context with low overhead [10, 34], rather than explicitly and blindly logging all call sites. We are also interested in leveraging aspects of data flow as well as control flow; analyses by Yuan et al. [43] 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. Future work could also consider aggregation of data from multiple failing runs in, for example, slice-based fault localization (e.g. [26]) or some form of union slicing (e.g. [28]). Finally, we believe our traced information holds great promise for assisting with failure recreation via symbolic execution.

Acknowledgments

Mark Chapman provided invaluable assistance with understanding and manipulating CodeSurfer-generated PDGs. This research was supported in part by DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0953478 and CCF-1217582; 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

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

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

[3] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 85–96.

[4] P. Anderson, T. Reps, and T. Teitelbaum, "Design and implementation of a fine-grained software inspection tool," *IEEE Trans. Softw. Eng.*, vol. 29, no. 8, pp. 721–733, Aug. 2003.

[5] T. Apiwattanapong and M. J. Harrold, "Selective path profiling," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '02. New York, NY, USA: ACM, 2002, pp. 35–42.

[6] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *ICSE (1)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 255–264.

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

[8] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel, "A formalisation of the relationship between forms of program slicing," *Sci. Comput. Program.*, vol. 62, no. 3, pp. 228–252, Oct. 2006.

[9] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 2, Apr. 2007.

[10] M. D. Bond and K. S. McKinley, "Probabilistic calling context," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 97–112.

[11] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '02. New York, NY, USA: ACM, 2002, pp. 2–9.

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

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

[14] B. Gauf and E. Dustin, "The case for automated software testing," *Journal of Software Technology*, vol. 10, no. 3, pp. 29–34, Oct. 2007.

[15] R. Gupta, M. L. Soffa, and J. Howard, "Hybrid slicing: integrating dynamic information with static analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 4, pp. 370–397, Oct. 1997.

[16] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Syst. J.*, vol. 41, no. 1, pp. 4–12, Jan. 2002.

[17] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 35–46.

[18] S. Horwitz, B. Liblit, and M. Polishchuk, "Better debugging via output tracing and callstack-sensitive slicing," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 7–19, Jan. 2010.

[19] C. Jaramillo, R. Gupta, and M. L. Soffa, "FULLDOC: A full reporting debugger for optimized code," in *Proceedings of the 7th International Symposium on Static Analysis*, ser. SAS '00. London, UK, UK: Springer-Verlag, 2000, pp. 240–259.

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

[21] M. Kamkar, P. Fritzson, and N. Shahmehri, "Three approaches to interprocedural dynamic slicing," *Microprocessing and Microprogramming*, vol. 38, no. 1-5, pp. 625–636, 1993.

[22] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988.

[23] ——, "Dynamic slicing of computer programs," *J. Syst. Softw.*, vol. 13, no. 3, pp. 187–195, Dec. 1990.

[24] J. Krinke, "Context-sensitivity matters, but context does not," in *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, ser. SCAM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 29–35.

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

[26] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," in *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference*, ser. COMPSAC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–10.

[27] D. Melski and T. W. Reps, "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*, ser. CC '99. London, UK, UK: Springer-Verlag, 1999, pp. 47–62.

[28] A. Mulhern and B. Liblit, "Effective slicing: A generalization of full and relevant slicing," University of Wisconsin–Madison, Tech. Rep. 1639, Jun. 2008.

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

[30] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: continuous evolution of software after deployment," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 65–69.

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

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

[33] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. (2006, Sep.) Software–artifact infrastructure repository. [Online]. Available: http://sir.unl.edu/portal/

[34] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 525–534.

[35] T. Takada, F. Ohata, and K. Inoue, "Dependence-cache slicing: A program slicing method using lightweight dynamic information," in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 169–.

[36] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[37] C. M. Tice, "Non-transparent debugging of optimized code," Ph.D. dissertation, EECS Department, University of California, Berkeley, Nov 1999. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/6232.html

[38] K. Vaswani, A. V. Nori, and T. M. Chilimbi, "Preferential path profiling: compactly numbering interesting paths," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 351–362.

[39] G. A. Venkatesh, "The semantic approach to program slicing," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 107–119.

[40] D. Weeratunge, X. Zhang, and S. Jagannathan, "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*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 155–166.

[41] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 352–357, Jul. 1984.

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

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

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

[45] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, ser. PLDI '04. New York, NY, USA: ACM, 2004, pp. 94–106.