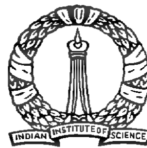
Efficient Whole Program Path Tracing

A Thesis
Submitted For the Degree of
Master of Science (Engineering)
in the Computer Science and Engineering

by

Sridhar G.



Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012

NOVEMBER 2018

© Sridhar G.

NOVEMBER 2018

All rights reserved

TO

My Family

Acknowledgements

First and foremost, I would like to thank my research advisor Prof. Murali Krishna Ramanathan for his invaluable support and guidance, without which this thesis would not have been possible. He has been very patient with my research and also very helpful in providing feedback about my work. I have learned a great deal working under him. I would like to thank my collaborator Prof. Suresh Jagannathan for his support throughout the course of this thesis. He has been influential in providing key insights about the problem statement, and discussions with him have been very helpful in improving the quality of work presented in this thesis.

I would also like to thank my lab mates at the Scalable Software Systems lab – Malavika, Monika, Nikita, and Subhendu for helping me get started in the initial phase of my research. Also, I thank all my friends including Akash, Aravind, Adarsh, Kumudha, Meghana, Vinay, for making my stay at IISc fun and enjoyable.

I would like to express my gratitude to N R Prashanth, for sparking my interest in computer science and also motivating me to pursue my graduate studies. I also thank my parents and my brother for their continuous love and support.

I would like to thank the office staff at the Department of CSA, for their efforts in making the administrative tasks run smoothly. I also want to thank the Ministry of Human Resource Development for supporting me with the scholarship.

Publications based on this Thesis

[Under submission] *Efficient Whole Program Path Tracing* **Sridhar Gopinath, Murali Krishna Ramanathan, Suresh Jagannathan** ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), Oct 2017.

Abstract

Obtaining an accurate whole program path (WPP) that captures a program's runtime behavior in terms of a control-flow trace has a number of well-known benefits, including opportunities for code optimization, bug detection, program analysis refinement, etc. Existing techniques to compute WPPs perform sub-optimal instrumentation resulting in significant space and time overheads. Our goal in this thesis is to minimize these overheads without losing precision.

To do so, we design a novel and scalable whole program analysis to determine instrumentation points used to obtain WPPs. Our approach is divided into three components: (a) an efficient summarization technique for inter-procedural path reconstruction, (b) specialized data structures called conflict sets that serve to effectively distinguish between pairs of paths, and (c) an instrumentation algorithm that computes the minimum number of edges to describe a path based on these conflict sets. We show that the overall problem is a variant of the minimum hitting set problem, which is NP-hard, and employ various sound approximation strategies to yield a practical solution.

We have implemented our approach and performed elaborate experimentation on Java programs from the DaCapo benchmark suite to demonstrate the efficacy of our approach across multiple dimensions. On average, our approach necessitates instrumenting only 9% of the total number of CFG edges in the program. The average runtime overhead incurred by our approach to collect WPPs is 1.97x, which is only 26% greater than the overhead induced by only instrumenting edges guaranteed to exist in an optimal solution. Furthermore, compared to the state-of-the-art, we

observe a reduction in runtime overhead by an average and maximum factor of 2.8 and 5.4, respectively.

Contents

Acknowledgements **i**

Publications based on this Thesis **ii**

Abstract **iii**

Keywords **ix**

1 Introduction **1**

2 Motivation **6**

3 Design Intuition **10**

4 Optimal Instrumentation **16**
4.1 Handling intersecting paths 16
4.2 Handling loops 18
4.3 Correctness and minimality 20

5 Optimizations for Conflict Set Generation **22**
5.1 Preliminaries 22
5.2 Dominance relation 23
5.3 Initial(α)-instrumentation 24

6 Modular Design **27**
6.1 Approximate(δ)-instrumentation 28
6.2 Function summaries 29
6.3 Conflict set generation 30
6.4 Instrumentation algorithm 32
6.5 Illustrative example 32

7 Implementation **35**

8 Experimental Evaluation **37**

9 Related Work **43**

10 Conclusions **46**

List of Tables

4.1 Generation of conflict sets 18

6.1 Whole program CFG *vs.* per function CFG. 28

8.1 Offline analyses time comparison. 42

List of Figures

1.1 Illustrative example 3

2.1 Motivating example. 7

3.1 Design challenges 14

4.1 Loop transformation. 19

5.1 α-instrumentation strategy. 24

6.1 δ-instrumentation strategy. 28
6.2 Summary generation. 29
6.3 Illustrative example. 33

8.1 Static instrumentation comparison. 38
8.2 Classification of instrumented edges. 39
8.3 Time overhead comparison 40
8.4 Space overhead comparison 41
8.5 Impact of α-instrumentation strategy. 42

Keywords

Program Tracing, Path Profiling, Program Analysis, Dynamic Control-Flow, Software Testing, Performance Analysis

Chapter 1

Introduction

Characterizing the dynamic behavior of a program in terms of its control-flow properties has been studied using a number of elegant techniques, including whole program path tracing [19; 39; 32], path profiling [3; 36; 13], call trace collection [37], calling context encoding [31; 7; 40], etc. The design trade-off in these techniques involves balancing runtime overhead with precision; an effective technique delivers *precise* information about an execution's control-flow with *low* runtime overhead.

Whole program path tracing is an important instance of these approaches that derives a complete control-flow trace of an execution. Doing so has multiple benefits to compilers, debuggers, and other related tools, including identifying opportunities for code optimization [25; 29], detecting bugs [10; 34; 26], aiding program analysis [20; 24; 1], inferring program properties [4], etc. More recently, intra-thread whole program paths have been used to enable debugging of concurrent programs by employing offline dynamic symbolic execution [14; 21; 22]. Beyond these applications, offline analyses on whole program paths can also be employed to extract path profiles, call traces, calling contexts, etc. whose information can be used to drive various optimizations. Hence, any efficient solution that obtains such path information serves as a critical component necessary for the efficient solution of a wide range of problems.

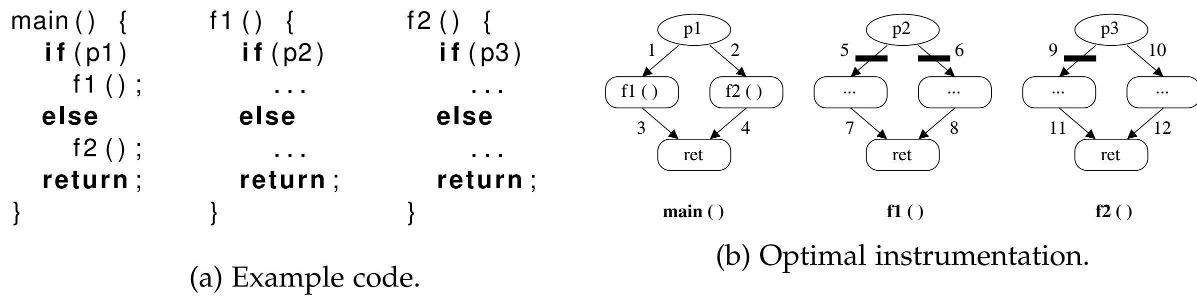
Whole program paths (WPPs) are obtained by instrumenting [15] certain program points in the program; runtime overhead is directly related to the number of such

instrumentation points and where they are inserted in the program. A naïve approach involves instrumenting outgoing edges of *all* branches present in the program's control-flow graph (CFG). While this strategy is easily shown to be correct, it is obviously not efficient, inserting many more instrumentations than necessary to accurately disambiguate different paths. An *optimal* instrumentation strategy inserts a minimum number of instrumentation points that nonetheless are sufficient to reconstruct the program's control-flow for a given execution. The number of such instrumentations should be minimal - the removal of any instrumentation point should prevent the unambiguous reconstruction of some feasible path. Numerous prior work have attempted to address this problem [28; 23; 18; 27; 2] using a variety of non-trivial, sophisticated solutions. For single procedure programs, the problem has been shown to be NP-complete [23; 2].

For example, Larus [19] presents a solution that divides WPPs into a number of segments, where each segment is an intra-procedural acyclic path. Segments are uniquely identified by employing the Ball and Larus [3] algorithm, which assigns a path identifier for that segment. A segment ends at a function call, return, or loop backedge. The program is then instrumented to emit the path identifier at the end of each segment. Upon executing this instrumented program, a log of emitted identifiers is generated, which is used to obtain the WPP.

While this is certainly an effective approach, there remain multiple opportunities to reduce the number of instrumentation points injected without sacrificing precision – (a) logging before each call site is *redundant*, since instrumented edges within a function can often be used to infer the execution of its call site, (b) logging at the end of each iteration of a loop is not always necessary, since the loop iteration can be precisely inferred by the path taken in the body of the loop, and (c) overall program structure is not leveraged. These observations lead us to consider the design of an optimal WPP instrumentation strategy, where a minimum number of instrumentation points are used to derive *any* whole program path in the program.

We illustrate the potential to reduce instrumentation points using the example



(a) Example code.

(b) Optimal instrumentation.

Figure 1.1: Illustrative example. Marked edges are instrumented. Presence of edge 5 or 6 in the log infers the execution of edge 1; presence of edge 9 or an *empty-log* infers the execution of edge 2.

given in Fig 1.1. The input program and its control-flow graphs are shown in Fig 1.1a and Fig 1.1b respectively. Fig 1.1b presents the outcome of an optimal instrumentation strategy in which marked edges are instrumented to emit the edge identifier, that can be subsequently used to identify the path taken. For example, if the log of an execution contains the edge labeled `5`, it implies the following path: (`p1` is true $\rightarrow$ `f1` is called $\rightarrow$ `p2` is true $\rightarrow$ `ret` from `f1`$\rightarrow$ `ret` from `main`). Similarly, the other three paths in the program are identified by the presence of either the edge identifiers `6` or `9`, or the absence of any entries in the log. Note that the strength of this instrumentation strategy is that, even though none of the edges in `main` is instrumented, a precise whole program path for any execution can still be obtained. By leveraging our understanding of overall program structure, we can use a small collection of instrumented edges to infer the execution of call sites and other edges in the program.

We propose a novel, scalable and effective instrumentation algorithm that takes as input a program $\mathcal{P}$ and outputs an instrumented program $\mathcal{P}^{\mathcal{I}}$, which contains edge instrumentations. On executing $\mathcal{P}^{\mathcal{I}}$, a log containing the emitted identifiers is generated. This log (a partial trace) is further processed offline using a regeneration algorithm to generate the executed whole program path.

Our approach is based on the following intuition - the ability to reconstruct a whole program path is tantamount to constructing a partition over edges comprising a program's control-flow graph. Each element in this partition is a set of edges (called

a *conflict set*), any one of which can serve as a representative in the reconstructed path; the number of instrumentations needed for path reconstruction is thus directly proportional to the number of sets in the partition. For example, edges from the true and false branch of a conditional naturally reside in the same set, since choosing any one of them is sufficient to recognize the specific branch taken by the execution. The efficiency of any algorithm based on this idea is thus dependent on the number of these constructed sets - our goal is to minimize the number of such sets, subject to the constraint that any reconstructed path using this partition precisely reflects the execution's control-flow.

However, applying this intuition to yield a practical solution is non-trivial. The inherent NP-hardness of constructing this partition, a variant of the minimum hitting set problem [12], coupled with structural and scalability challenges apparent when analyzing large real-world applications, conspire against a straightforward implementation of this idea. The focus of this thesis is centered on overcoming this hurdle.

We have implemented our ideas on top of the SOOT Java compilation framework [35]. Experiments conducted on a number of DACAPO benchmarks [5] show that on average instrumenting *only* 9% of the overall edges present in the program is sufficient to derive WPPs. More importantly, our experimental results reveal that an optimal strategy would incur at least 71% runtime overhead on average over these benchmarks, albeit with analysis times reflecting the inherent NP-hardness of the problem; in contrast, our approach incurs 97% overhead, but with small analysis time overheads. Moreover, a comparison with Larus [19] shows significant performance gains of our technique – up to 5.4x on average compared to the state-of-the-art. Reasonably low runtime overhead, the ability to derive precise WPPs, negligible analysis time to identify instrumentation points, and demonstrated applicability of our approach to realistic Java programs, make our implementation a compelling tool for practical adoption.

The thesis makes the following technical contributions:

- We formulate the problem of finding the minimum number of instrumentation points to derive WPPs as a variant of the minimum hitting set problem.
- We propose a novel approach, that addresses challenges pertaining to loops and function calls to efficiently identify the instrumentation points.
- We provide a lower bound on the runtime overhead incurred by any approach that corresponds to a feasible instrumentation strategy to derive WPPs.
- We present a scalable and effective design and demonstrate with elaborate experimentation that WPPs can be obtained with a relatively small number of instrumentations.

The rest of the thesis is organized as follows – Chapter 2 motivates the proposed approach for WPPs using an example. We provide the basic design intuition and demonstrate the similarity to minimum hitting set in Chapter 3. We extend the design to real programs and propose optimizations, assuming the feasibility of path enumeration, in Chapters 4 and 5. In Chapter 6, we propose a modular design to overcome the challenges with path enumeration in real programs and address Java-specific implementation details in Chapter 7. We present the experimental results in Chapter 8 and discuss related work in Chapter 9 before concluding in Chapter 10.

Chapter 2

Motivation

Fig 2.1a presents two Java classes taken from `luindex`, a member of the DaCapo [5] benchmark suite. Methods `A`, `B` and `D` from the `ConcurrentMergeScheduler` class and method `C` from `IndexWriter` class are shown. The method definitions, names and line numbers are simplified for ease of presentation. Fig 2.1b presents the corresponding CFGs, where statements in each node are represented by their corresponding line numbers. Table 2.1c shows the instrumentation obtained by applying the following three approaches:

- *WPP-L [19]*: WPP-L splits the whole program paths into a number of intra-procedural acyclic paths. Each acyclic path is then uniquely identified using the Ball and Larus [3] numbering scheme to help identify the path executed.
- *Optimal instrumentation strategy*: This strategy utilizes program structure to find a minimum number of program points, such that instrumenting these points is sufficient to identify any whole program path. In Chapter 3, we show that this strategy is NP-hard.
- *Our approach*: The approach presented in this thesis is based on the optimal instrumentation strategy. However, since obtaining the optimal instrumentation is NP-hard, we employ several novel approximations to yield a practical and scalable solution.

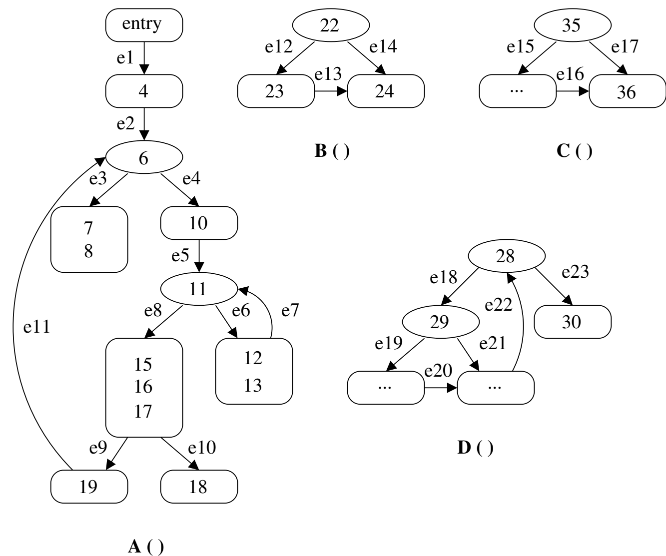
```
ConcurrentMergeScheduler.java:
1 public class ConcurrentMergeScheduler
     extends ... {
2  protected IndexWriter writer;
3  public void A(IndexWriter writer) throws
     ... {
4   B(...);
5   while(true) {
6    if (merge == null) {
7     B(...);
8     return;
9    }
10   int x = D();
11   while (x >= maxThreadCount) {
12    B(...);
13    x = D();
14   }
15   B(...);
16   x = D();
17   if ( x >= maxThreadCount)
18    return;
19   B(...);
20 } }

21 private void B(String msg) {
22  if (writer != null)
23   writer.C(msg);
24  return;
25 }

26 private synchronized int D() {
27  int count = 0;
28  for(int i = 0; i < mergeThreads.size();
     i++ )
29   if (...) ...
30  return count;
31 } }

IndexWriter.java:
32 public class IndexWriter {
33  private PrintStream info;
34  public void C(String msg) {
35   if (info != null) ...
36   return;
37 } }
```

(a) Code from benchmark.



(b) Control flow graphs.

Methods	WPP-L	Optimal	Our Approach
A ( )	E: e2, e3, e5, e7, e8, e9, e11 C: 4, 7, 10, 15, 12, 19 R: 18	E: e8	E: e8, e10
B ( )	E: e12, e13 C: 23 R: 24	E: e14	E: e14
C ( )	E: e15 R: 36	E: e15, e17	E: e15, e17
D ( )	E: e18, e19, e22 R: 30	E: e18, e19, e23	E: e18, e19, e23
Total	**24**	**7**	**8**

(c) Instrumentation points.

Figure 2.1: Motivating example.

Instrumentations on edges, call sites and return sites are represented by the labels `E`, `C` and `R` respectively. The instrumentation associated with the optimal strategy is obtained manually.

The WPP-L approach inserts a total of 24 instrumentations as shown in Table 2.1c. The optimal instrumentation strategy leads to only 7 instrumented edges, which is more than a 3x improvement over WPP-L. This disparity in instrumentation count motivates the design of an efficient instrumentation strategy. Note that our technique yields a solution close to the optimal, requiring only 8 edges to be instrumented.

Call sites WPP-L adds instrumentation to emit the BALL-LARUS identifier at the first call site in each basic block to record the acyclic path taken before entering the function. For the example program, seven call sites are instrumented as shown in Table 2.1c. We observe that some of these instrumentations are redundant. For example, the call site at line `23` to `C()` in `B()` is instrumented to record the branch at line `22`. But the outcome of the branch can be inferred using the instrumented edges in `C()` and the instrumented return site at line `24`. More specifically, if the log contains `e15` or the return site `36` due to the instrumentations in `C()`, we can infer the execution of edge `e12`. Similarly, the presence of return site `24` in the log can be used to infer the execution of edge `e14`. This makes the instrumentation at the call site `23` in `B()` unnecessary. A similar argument can be applied to the call sites at lines `4`, `7`, `10`, `19`.

Loop backedges In the presence of loops, the WPP-L approach instruments back edges to record the acyclic path taken in the body of the loop. But, observe that the information recorded in the loop body can be used to infer a loop iteration. For example, the loop represented by the back edge `e7` consists of a single path `e6` $\rightarrow$ `12` $\rightarrow$ `13` $\rightarrow$ `e7`, where the call site at `12` and edge `e7` are instrumented. Interestingly, we observe that the instrumentation in `B()` alone is sufficient to identify this path. This shows that instrumentations on the back edge of a loop are not always necessary and can introduce avoidable runtime overhead. Similar reasoning applies to the back

edges `e11` and `e22`.

In general, we can reduce the number of injected instrumentations by exploiting details about the program structure reflected in the CFG. However, due to the NP-hardness of the problem (discussed in Chapters 3 to 5) and the complexities associated with obtaining an optimal solution on real (large) programs (discussed in Chapter 6), we employ approximations that result in eight (rather than the optimal seven) instrumentation points shown in Table 2.1c. This difference in the number of instrumentation points (8 *vs* the 24 produced by WPP-L) has a corresponding beneficial impact on runtime overhead.

Chapter 3

Design Intuition

In this chapter, we define the problem of optimally instrumenting a program to obtain WPPs and show that this problem is a variant of the minimum hitting set problem.

Given a program $\mathcal{P}$, each function `F` in $\mathcal{P}$ is represented in terms of its CFG, where each node represents a statement and each directed edge represents control-flow from one statement to another. `F` is assumed to have a unique entry node ($entry_F$), a unique exit node ($exit_F$), and a unique exit edge. Edges between the CFGs associated with different functions exist in the presence of call sites. For a call site node `u` to a function G, a call edge is present from `u` to $entry_G$ and a return edge is present from $exit_G$ to `u`. The execution of the program starts from the `main` function. A *whole program path* is a sequence of edges in the program starting at $entry_{main}$ and ending at $exit_{main}$, representing a *valid* execution path for the program. Thus, a whole program path entering a call site will continue at the callee function. Given nodes n_1 and n_2 in a whole program path p, a program *path* starting at n_1 and ending at n_2 is a sequence of edges found in p that enables control flow from n_1 to reach n_2. A *partial* whole program path is a sub-sequence of a whole program path. We define two paths `L` and `R` to be *disjoint* if there is no common edge between them. Otherwise, `L` and `R` are considered to be *intersecting*.

Problem statement Given a program $\mathcal{P}$, the problem is to find the minimum set of instrumentation points $\mathcal{I}$ sufficient to generate an instrumented program $\mathcal{P}^{\mathcal{I}}$, where each instrumentation point emits a unique identifier when executed. Upon the execution of $\mathcal{P}^{\mathcal{I}}$, a sequence of identifiers $\mathcal{L}$ is generated, which represents a partial whole program path. $\mathcal{I}$ should satisfy the property that the complete whole program path can be generated *precisely* by using *only* $\mathcal{L}$ and $\mathcal{P}^{\mathcal{I}}$ with the help of an offline reconstruction algorithm.

Criteria for feasible solution A set $\mathcal{I}$ is a feasible solution to the program $\mathcal{P}$ if the partial log $\mathcal{L}$ can be used to generate the whole program path without any ambiguity i.e. $\mathcal{L}$ is sufficient to infer the edge taken at each branch encountered by the whole program path. For each branch node, $\mathcal{I}$ should be able to precisely distinguish between all the paths starting at that node. This subsequently guarantees that upon execution, $\mathcal{I}$ can precisely identify the branch edge executed. Thus, the criteria for a feasible solution becomes: for each branch η in $\mathcal{P}$, $\mathcal{I}$ should be able to distinguish between each pair of paths starting at η.

Definition 1 (Optimal solution)**.** *The optimality of a feasible solution is defined on the cardinality of $\mathcal{I}$. Given a program $\mathcal{P}$, a feasible solution $\mathcal{I}$ to $\mathcal{P}$ is said to be an* optimal solution*, if there exists no other feasible solution $\mathcal{I}'$ to $\mathcal{P}$ such that $\left|\mathcal{I}'\right| < |\mathcal{I}|$.*

We now discuss the idea behind obtaining the optimal solution for a program. For a branch node η, let `L` and `R` be two paths starting at η from the true and false edge of η respectively. For now, assume that `L` and `R` are disjoint[1]. Let `e` be an edge in path `L`. Instrumenting `e` is sufficient to distinguish between `L` and `R`, because the presence of `e` in $\mathcal{L}$ identifies `L` and its absence identifies `R`. Since the paths are disjoint, similar behavior is observed by picking any edge in `L` or any edge in `R`. If a set `C` contains all the edges in the path `L` and `R`, then any edge in `C` can be instrumented to distinguish between `L` and `R` at η. We define `C` as the *conflict set* for the pair of paths `L` and `R`.

[1]We relax this assumption in the following chapter, and explain the applicability of the approach to real programs.

Formally, for a branch node η, consider two whole program paths p1 and p2, where p1 and p2 reach η by traversing the same sequence of edges, diverge at η and reach the end of the program. Let $\texttt{p1} = (a_1 \ldots a_k \ldots a_m)$, $\texttt{p2} = (b_1 \ldots b_k \ldots b_n)$, such that $\forall i < k$, $a_i = b_i$, $a_k \neq b_k$, where a_k and b_k are the outgoing edges of η. Using p1 and p2, a pair of paths (sub-sequence of whole program paths) starting at η is derived as $\texttt{L} = (a_k \ldots a_m)$ and $\texttt{R} = (b_k \ldots b_n)$. Here, L signifies the path from the left edge of η and R signifies the path from the right edge of η. Thus, L and R are two paths starting from different outgoing edges of η. The instrumentation strategy should be able to distinguish between the pair of paths L and R. Since multiple such pairs are possible at η, we define $\texttt{pairs}(\eta)$ to generate all possible pairs. Thus, for each branch η, we need to be able to distinguish between each pair of paths in $\texttt{pairs}(\eta)$.

For any branch η, let us assume that each pair of paths in $\texttt{pairs}(\eta)$ is disjoint. Conflict sets are then collected for each pair of paths at each branch node, and the collection of conflict sets (call it $\mathbb{C}$) is then used to construct the set of instrumentation points $\mathcal{I}$. A feasible solution $\mathcal{I}$ should satisfy the property that $\forall \texttt{C} \in \mathbb{C}, \exists \texttt{e} \in \mathcal{I} \mid \texttt{e} \in \texttt{C}$ i.e. $\mathcal{I}$ should cover each conflict set. The minimum $\mathcal{I}$ that satisfies this property constitutes the minimum number of instrumentation points required for the given program. This formulation can be seen as a variant of the minimum hitting set problem [12].

Minimum hitting set problem Given a universe $\mathbb{U}$ of elements and a collection $\mathbb{C}$ of subsets of $\mathbb{U}$, we wish to find the smallest subset $\mathcal{H}$ of $\mathbb{U}$, which satisfies the property that $\mathcal{H}$ contains at least one element from each set in the collection $\mathbb{C}$. The minimum hitting set for $\mathbb{C}$ is $\mathcal{H}$. In our problem, $\mathbb{U}$ is the set of all edges and $\mathbb{C}$ is the collection of conflict sets for the program. Given this, the minimum hitting set $\mathcal{H}$ of $\mathbb{C}$ is the set of instrumentation points $\mathcal{I}$. Thus, after collecting conflict sets, any off-the-shelf hitting set solver can be employed to obtain $\mathcal{I}$.

Algorithm 1 presents the pseudo-code to generate the instrumented program $\mathcal{P}^{\mathcal{I}}$ for an input program $\mathcal{P}$. The MAIN procedure invokes GENCONFLICTSETS procedure

Algorithm 1 INSTRUMENT

Input: Program $\mathcal{P}$
Output: Set $\mathcal{I}$, instrumented program $\mathcal{P}^{\mathcal{I}}$

```
1: procedure MAIN(P)
2:     I := ∅, ℂ := ∅
3:     for function F in P do
4:         for branch η in F do
5:             GENCONFLICTSETS(η, ℂ)
6:     I := Minimum hitting set of ℂ
7:     P^I := P instrumented with edges in I

8: procedure GENCONFLICTSETS(η, ℂ)
9:     for paths (L, R) in pairs(η) do
10:        C := { Edges in L } ∪ { Edges in R }
11:        Add C to ℂ
```

for each branch node η in the program. The GENCONFLICTSETS procedure creates a conflict set for each pair of paths in `pairs(`η`)`. Since each pair of paths is assumed to be disjoint, the conflict set consists of edges in the two paths.

Algorithm 2 RECONSTRUCT

Input: Log $\mathcal{L}$, $\mathcal{P}^{\mathcal{I}}$
Output: Complete trace

```
1: trace := ∅, l := next(L), u := entry_main
2: while u ≠ exit_main do
3:     if outdegree(u) == 1 then v := succ(u)
4:     else
5:         L := unique path originating at u containing l
6:         v := succ(u) in L
7:     if (u → v) == l then l := next(L)          ▷ Consume log entry
8:     Add u → v to trace                          ▷ Record the executed edge
9:     u := v
```

Algorithm 2 presents the pseudo-code to reconstruct the executed whole program path using the generated log $\mathcal{L}$. The algorithm traverses the CFG of the instrumented program $\mathcal{P}^{\mathcal{I}}$ and makes decisions on the path taken using $\mathcal{L}$. On encountering a branch node, the current entry in the log is used to determine the branch edge to be traversed. If the instrumented edge representing the current log entry is encountered,

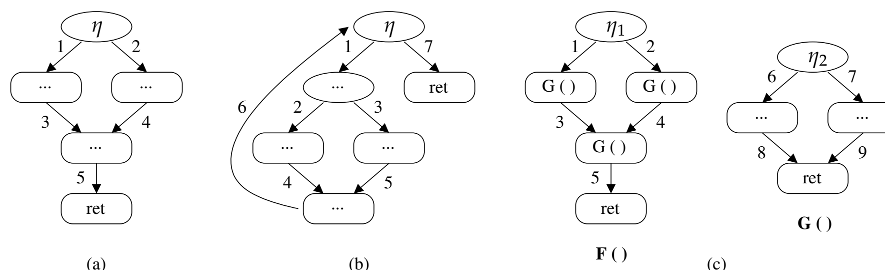


Figure 3.1: Challenges: (a) Intersecting paths. (b) Loops. (c) Number of paths.

the current log entry is consumed and the next entry in $\mathcal{L}$ is read.

Even though the presented approach is correct, there are challenges in applying it in practice – (a) the approach is proposed in the context of disjoint paths and does not handle intersecting paths which are common in programs, (b) cycles in the program result in an infinite number of paths and (c) path enumeration is not practical. We now explain these challenges in more detail.

Intersecting paths Instrumenting common edges found among intersecting paths leads to ambiguity during reconstruction. For example, consider the program in Fig 3.1(a). The two paths `L` and `R` are $(1,3,5)$ and $(2,4,5)$. The conflict set created for `L` and `R` is $\texttt{C} = \{1,3,5,2,4\}$. Since `C` is the only conflict set, a possible minimum hitting set is $\mathcal{I} = \{5\}$. If the edge `5` is instrumented, the presence of `5` in the log will not be sufficient to distinguish between the paths `L` and `R`, because `5` is common in both `L` and `R`. In such a case, instrumentation on non-intersecting edges is required to distinguish the two paths.

Loops In the presence of loops, the number of paths starting at a branch will become unbounded. This is illustrated in Fig 3.1(b). At the branch node η, the paths $(1,2,4,6,7)$ and $(1,2,4,6,1,3,5,6,7)$ represent paths with one and two iterations of the loop, respectively. Continuing this way, the number of paths starting at η, and the conflict sets created for each pair of paths at η, will become unbounded. In these cases, a mechanism which overcomes this challenge to identify the minimum instrumentation points is needed.

Enumerating paths Conflict sets are created by enumerating each pair of paths starting at branch nodes. The number of such pairs grows exponentially with program size. Consider the simple program in Fig 3.1(c), which contains three call sites in the function `F` to the function `G`. At the branch node η_1 in `F`, 16 pairs of paths are feasible and at the branch node η_2 in `G`, 9 pairs of paths are feasible. This results in the generation of 25 conflict sets for the simple program. As the program size increases, the number of conflict sets created also increases, and the enumeration of each pair of paths becomes practically infeasible.

Chapter 4

Optimal Instrumentation

In this chapter, we discuss strategies to handle intersecting paths and cyclic CFGs in real programs designed to yield minimal (and correct) instrumentation.

4.1 Handling intersecting paths

Paths L and R are said to be intersecting if there is a common edge between them i.e. $\exists e \mid (e \in L) \wedge (e \in R)$. Here, e is called the intersecting edge. Creating a conflict set containing an intersecting edge is incorrect because instrumentation affixed to that edge will not be able to distinguish between the two paths. To resolve this ambiguity, a conflict set C_e, consisting of edges in the paths L and R up to e is created. Each edge in C_e is unique to either L or R up to e, and instrumenting any edge in C_e is sufficient to distinguish between the paths L and R.

In the presence of multiple intersecting edges, ambiguity must be resolved for each edge. This is because any edge may be encountered during an arbitrary execution and if sufficient instrumentation is not present, then the executed path becomes ambiguous. Hence, a conflict set is created for each intersecting edge. In some paths, multiple instances of the edge e can appear. For example, if a path L consists of two call sites to the same function F, the edges in F an appear twice in L. Let e_1 and e_2 be two instances of an intersecting edge e in path L. In this case, the conflict set created

for e_2 will contain the edge e, since e_1 is present in the path up to e_2. However, instrumenting the edge e to cover the conflict set is incorrect. This is because e is not unique to either L or R, and fails to distinguish the two paths. Thus, a conflict set created for an intersecting edge e, should not contain e. Due to this reason, only the first instance of e is considered while creating the conflict set.

For ease of presentation, we define two auxiliary functions to describe our strategy:

- `index(e,L)`: the index of the first instance of e in L.
- `edges(e,L)`: a set, consisting of edges in the path L up to the index k, where k = `index(e,L)`.

For a pair of paths L and R, we derive the set of intersecting edges, denoted as $IE_{L,R}$, and create a conflict set for each intersecting edge e in $IE_{L,R}$ as C_e = `edges(e,L)` $\cup$ `edges(e,R)`.

We elaborate this further using a simple example. Let L $= (1, 7, 3, 5, 6, 7)$ and R $= (2, 4, 5, 6, 7)$ be two paths starting at a branch node. Then, $IE_{L,R} = \{7, 5, 6\}$ and conflict sets are created for each edge in $IE_{L,R}$. Let us consider the intersecting edge 7. The `edges(7,L)` is $\{1\}$. Similarly, for path R, `edges(7,R)` is $\{2, 4, 5, 6\}$. The conflict set created for edge 7 is $C_7 = \{1, 2, 4, 5, 6\}$. Here, even though the edge 7 appears twice in L, we considered only the first instance when constructing C_7. Similarly, conflict sets for edges 5 and 6 are constructed as $C_5 = \{1, 7, 3, 2, 4\}$ and $C_6 = \{1, 7, 3, 5, 2, 4\}$.

If L and R do not contain any intersecting edges, a unique element $\perp$ is added to $IE_{L,R}$, which represents the end of the paths. In this case, the conflict set will contain all the edges in the paths L and R.

Algorithm 3 presents the pseudo-code to handle intersecting paths. Since the MAIN procedure from Algorithm 1 invoking GENCONFLICTSETS remains the same, it has been omitted. For each pair of paths (L, R) in `pairs`(η), conflict set is created for each edge in $IE_{L,R}$. The collected conflict sets are further used to identify the instrumentation points.

Algorithm 3 INSTRUMENT

```
1: procedure GenConflictSets(η, ℂ)
2:     for paths (L, R) in pairs(η) do
3:         Compute IE_{L,R}
4:         for e in IE_{L,R} do
5:             C := edges(e,L) ∪ edges(e,R)
6:             Add C to ℂ
```

η	pairs(η)		$IE_{L,R}$	C
	L	R		
p2	(5,7,3)	(6,8,3)	{3}	$C_3 = \{5,7,6,8\}$
p3	(9,11,4)	(10,12,4)	{4}	$C_4 = \{9,11,10,12\}$
p1	(1,5,7,3)	(2,9,11,4)	{⊥}	$C_\perp = \{1,5,7,3,2,9,11,4\}$
	(1,5,7,3)	(2,10,12,4)	{⊥}	$C_\perp = \{1,5,7,3,2,10,12,4\}$
	(1,6,8,3)	(2,9,11,4)	{⊥}	$C_\perp = \{1,6,8,3,2,9,11,4\}$
	(1,6,8,3)	(2,10,12,4)	{⊥}	$C_\perp = \{1,6,8,3,2,10,12,4\}$

Table 4.1: Algorithm 3 on Fig 1.1.

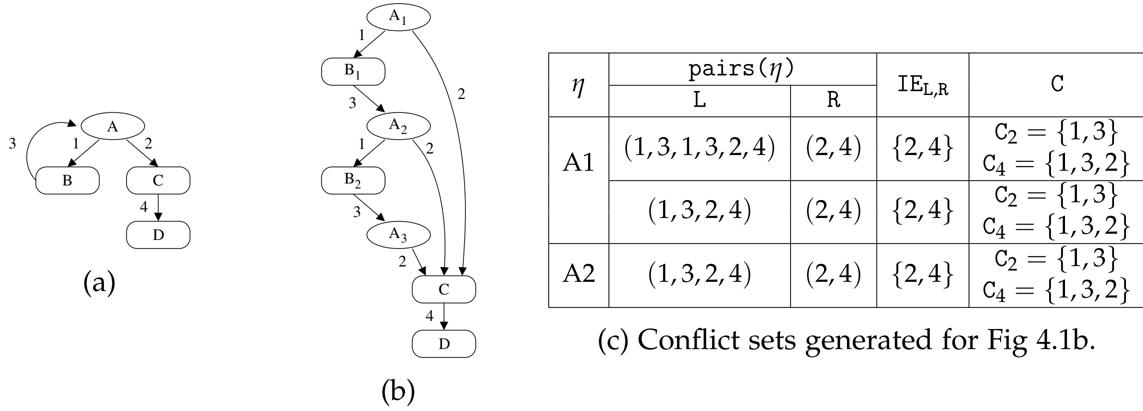
Illustrative example We now explain Algorithm 3 on the illustrative example given in Fig 1.1. Table 4.1 shows the various information collected by the algorithm to create conflict sets. For the branch nodes p3, p2 and p1, the numbers of pairs of paths present in pairs(η) are 1, 1 and 4 respectively. These paths are shown in Table 4.1, and a conflict set is created for each pair, as there is a maximum of one intersecting edge for all pairs of paths. Then, the minimum hitting set is computed for the six conflict sets. A possible optimal solution to cover the six conflict sets is $\{5, 6, 9\}$. These edges are represented as marked edges in Fig 1.1b.

4.2 Handling loops

We discuss the optimal approach to instrument programs with cyclic CFGs. In the presence of loops in a function, the number of paths becomes unbounded and can thus lead to the generation of infinite number of conflict sets.

A naïve approach to handle loops involves instrumenting the backedges of the loop, and removing the instrumented backedges to make the CFG acyclic. But instrumenting all the backedges is not optimal always. Another approach is to unroll the loop depending on the iteration count of the loop. Since the iteration count is not available statically, this approach is also not practical.

To address this, we make a key observation about the behavior of the loop across multiple iterations. The static body of the loop across each iteration of the loop is identical even though the values of variables can be different. Since path tracing requires only the control-flow structure, we can ignore the changes to variables. This



η	pairs(η) L	pairs(η) R	$IE_{L,R}$	C
A1	(1,3,1,3,2,4)	(2,4)	{2,4}	$C_2 = \{1,3\}$ $C_4 = \{1,3,2\}$
	(1,3,2,4)	(2,4)	{2,4}	$C_2 = \{1,3\}$ $C_4 = \{1,3,2\}$
A2	(1,3,2,4)	(2,4)	{2,4}	$C_2 = \{1,3\}$ $C_4 = \{1,3,2\}$

(c) Conflict sets generated for Fig 4.1b.

Figure 4.1: Loop transformation.

enables us to simply unroll the loop once and apply Algorithm 3 on the resulting acyclic CFG without affecting minimality or correctness. More formally,

Property 1. *If any path taken in two iterations of the loop can be distinguished without ambiguity, it guarantees that the path taken in n iterations of the loop can also be distinguished, for an arbitrary value of n.*

Proof. We prove by induction on the number of loop iterations. Let us assume that we have an approach to find the instrumentation points to identify the path taken in the *first* and *second* iterations of the loop. As induction hypothesis, assume that the instrumented edges are able to identify the path taken till k iterations, for any value of k. We have to prove that the path taken in $k+1$ iteration can also be identified. As explained previously, the body of the loop is identical for all iterations of the loop. Hence, the instrumentation points required to identify the path in k and $k+1$ iterations are equal to the instrumentation points required for the *first* and *second* iterations. Since an approach to find the latter is assumed to exist, the same solution is sufficient for the former as well. Thus, the path taken in $k+1$ iteration can be identified, proving our induction hypothesis. □

Our strategy to identify paths in two iterations is to unroll the loop once in the CFG. This results in an acyclic graph with two identical copies of the loop body, and we use the same edge identifiers in both copies. We elaborate this using a simple

example. Fig 4.1a shows a CFG with a loop. After applying our transformation, the unrolled loop is shown in Fig 4.1b. Block B_2 is identical to the block B_1, and the branches A_2 and A_3 are identical to the branch A_1. Since we only consider two replications of the body of the loop, the outgoing edge 1 of the branch A_3 is omitted. Hence, A_3 is treated as a primitive statement and not as a branch node. Algorithm 3 is applied on this transformed CFG to obtain the instrumentation points. Using this, edges are instrumented on the original cyclic CFG.

Let us assume that the body of the loop B contains a single statement. Conflict sets are collected on the acyclic graph shown in Fig 4.1b at the branch nodes A_1 and A_2. The branch A_3 is not considered since it has only one outgoing edge. The numbers of pairs of paths starting at A_1 and A_2 are 2 and 1 respectively. A total of six conflict sets are collected for these pairs as shown in Fig 4.1c. Consider the paths $\texttt{L} = \{1, 3, 2, 4\}$ and $\texttt{R} = \{2, 4\}$ at A_1. Here, 2 and 4 are the intersecting edges i.e. $\texttt{IE}_{\texttt{L,R}} = \{2, 4\}$. Hence, two conflict sets $\texttt{C}_2$ and $\texttt{C}_4$ are created. A minimum hitting set for the six conflict sets is $\{1\}$. Edge 1 is then instrumented in the cyclic CFG in Fig 4.1a, which is capable of identifying any WPP.

4.3 Correctness and minimality

We now prove the correctness and minimality of the instrumentation points obtained using this strategy.

Theorem 1 (Correctness)**.** *Given a program $\mathcal{P}$, the set of instrumentation points $\mathcal{I}$ obtained using Algorithm 3 is sufficient to regenerate any path in the program, using Algorithm 2.*

Proof. We first prove the theorem for programs with acyclic CFGs. Assume that during reconstruction at a branch η, there is ambiguity between two paths `L` and `R`. This indicates that the next entry in the log $\mathcal{L}$ generated by an edge `e` is unable to distinguish between `L` and `R` i.e. `e` is not unique to either `L` or `R`. Since `e` is the next instrumented edge, none of the edges to `e` is instrumented in both paths. Here, `L` and `R` are intersecting paths and `e` is an intersecting edge i.e. $\texttt{e} \in \texttt{IE}_{\texttt{L,R}}$

In Algorithm 3, a conflict set is created for each edge in $\texttt{IE}_{\texttt{L,R}}$. The conflict set $\texttt{C}_\texttt{e}$ created for the edge e consists of edges such that instrumenting any edge in $\texttt{C}_\texttt{e}$ can distinguish between L and R. Since none of the edges till e is instrumented, it indicates that none of the edges in $\texttt{C}_\texttt{e}$ is instrumented i.e. $\mathcal{I}$ does not cover $\texttt{C}_\texttt{e}$. This contradicts the fact that $\mathcal{I}$ is a feasible solution, since it does not satisfy the hitting set property. This proves the correctness of our approach for acyclic CFGs.

For programs with cyclic CFGs, we employ the loop transformation described earlier to convert the cyclic CFG to an acyclic CFG, which satisfies Property 1. Since the correctness of Algorithm 3 is known for acyclic CFGs, the transformation enables it to be generalized for cyclic CFGs. □

Theorem 2 (Minimality)**.** *Given a program $\mathcal{P}$, the cardinality of an optimal solution $\mathcal{O}$ (from Definition 1) is equal to the cardinality of $\mathcal{I}$ obtained using Algorithm 3.*

Proof. Let $\mathbb{C}$ be the collection of conflict sets. Since $\mathcal{O}$ is defined to be the minimum set, the size of any other feasible solution will be greater than or equal to $\mathcal{O}$ i.e. $|\mathcal{O}| \leq |\mathcal{I}|$. We first show that $\mathcal{O}$ also satisfies the hitting set property for $\mathbb{C}$. Consider a conflict set $\texttt{C} \in \mathbb{C}$ for a pair of paths L and R such that C is not covered by $\mathcal{O}$. Since none of the edges in C is instrumented, there exists an execution where $\mathcal{O}$ will not be able to distinguish between L and R. This will make $\mathcal{O}$ infeasible. As $\mathcal{O}$ is assumed to be a feasible solution, no such C can exist. Thus, each set in $\mathbb{C}$ contains at least one edge in $\mathcal{O}$ i.e. $\mathcal{O}$ is a hitting set for $\mathbb{C}$. Since $\mathcal{I}$ is defined to be the minimum hitting set, the size of any other hitting set to $\mathbb{C}$ cannot be less than that of $\mathcal{I}$ i.e. $|\mathcal{I}| \leq |\mathcal{O}|$. As we already know that $|\mathcal{O}| \leq |\mathcal{I}|$, this proves that $|\mathcal{O}| = |\mathcal{I}|$. □

Chapter 5

Optimizations for Conflict Set Generation

The complexity of deriving instrumentation points is dependent on the total number of conflict sets generated. In this chapter, we discuss two optimizations to reduce the total count of generated conflict sets. Importantly, this reduction will not affect the correctness and minimality associated with the overall approach.

5.1 Preliminaries

We first define two interesting properties related to the solution for the hitting set problem used in our approach.

Property 2. *For a universe of elements* $(\mathbb{U})$, *and a collection of subsets of* $\mathbb{U}$, $(\mathbb{C})$, *if* A *and* B *are two sets in* $\mathbb{C}$ *such that* $A \subset B$, *then the minimum hitting set of* $\mathbb{C}$ *is equal to the minimum hitting set of* $\mathbb{C} - B$.

Property 3. *For a universe of elements* $(\mathbb{U})$, *and a collection of subsets of* $\mathbb{U}$, $(\mathbb{C})$, *let* $\mathcal{H}$ *be* $\mathbb{C}$*'s minimum hitting set. If* $S \subseteq \mathcal{H}$, *and* $\mathbb{C}' = \mathbb{C}/S$, *then* $\mathcal{H} - S$ *is a minimum hitting set of* $\mathbb{C}'$.[1]

[1] $\mathbb{C}/S$ *is defined as: Given* $\mathbb{C} = \{A_1, \ldots, A_n\}$, $A_i \in \mathbb{C}/S$ *if* $A_i \cap S = \emptyset$.

The first property captures the scenario where $\{A, B\} \subseteq \mathbb{C}$ and $A \subset B$. In this case, B can be ignored while finding the hitting set of $\mathbb{C}$. The second property describes the scenario where a set S contains some elements of $\mathbb{U}$ before finding the hitting set of a collection of sets $\mathbb{C}$. All the sets in $\mathbb{C}$ containing an element from S can be ignored, since they already satisfy the hitting set property. To illustrate this, consider a set $A = \{1, 2, 3\}$ in a collection of sets $\mathbb{C}$. Let S be initialized with some elements, $S = \{1\}$. Here, the element `1` in S is also present in the set A, satisfying the hitting set property for A.

5.2 Dominance relation

We observe that it is not necessary to construct a conflict set for *each* intersecting edge in Algorithm 3. Our observation is inspired by Property 2.

For a pair of paths `L` and `R`, we define a dominance relation, $\prec_{\mathtt{L},\mathtt{R}}$, over pairs of intersecting edges. An intersecting edge `e` *dominates* another intersecting edge `f` ($\mathtt{e} \prec_{\mathtt{L},\mathtt{R}} \mathtt{f}$), if `e` appears before `f` in both the paths `L` and `R`. In other words, $\mathtt{index(e,L)} < \mathtt{index(f,L)}$ and $\mathtt{index(e,R)} < \mathtt{index(f,R)}$ are both true.

Consider two intersecting edges `e` and `f` such that $\mathtt{e} \prec_{\mathtt{L},\mathtt{R}} \mathtt{f}$. Here, $\mathtt{edges(e,L)} \subset \mathtt{edges(f,L)}$ and $\mathtt{edges(e,R)} \subset \mathtt{edges(f,R)}$. This implies that the conflict set created for `e` is a subset of the conflict set created for `f`, $\mathtt{C_e} \subset \mathtt{C_f}$. From Property 2, $\mathtt{C_f}$ will have no effect on the minimum hitting set. Hence, $\mathtt{C_f}$ can be ignored. By extending this property, conflict sets for all the intersecting edges dominated by `e` are also ignored.

For example, suppose $\mathtt{L} = (1, 7, 3, 5, 6, 7)$ and $\mathtt{R} = (2, 4, 5, 6, 7)$. The set of intersecting edges is $\mathtt{IE}_{\mathtt{L},\mathtt{R}} = \{7, 5, 6\}$. Here, the edge 5 dominates the edge 6, and thus $5 \prec_{\mathtt{L},\mathtt{R}} 6$. Hence, $\mathtt{C}_5 \subset \mathtt{C}_6$. Consequently, $\mathtt{C}_6$ can be ignored. However, we still need to construct the conflict set $\mathtt{C}_7$ as there is no dominance relation between the edges 5 and 7. Based on this, we use $\mathtt{C}_5$ and $\mathtt{C}_7$ to distinguish between `L` and `R`.

By leveraging the above property, for any pair of paths, conflict sets are only created for the intersecting edges which are *not* dominated by any other edge for

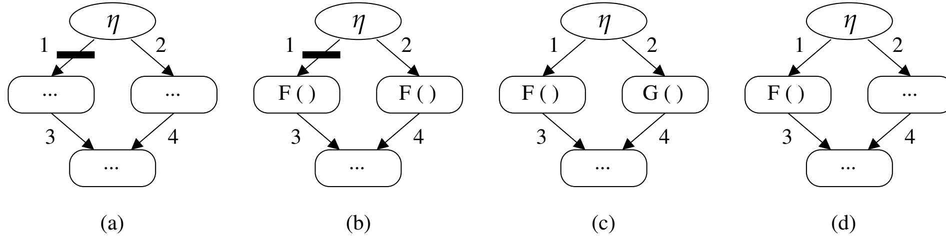


Figure 5.1: α-instrumentation strategy.

that pair. We redefine $\mathtt{IE}_{L,R}$ to contain only these edges. Thus, for a pair of paths L and R, and the set $\mathtt{IE}_{L,R}$, each edge f in $\mathtt{IE}_{L,R}$ should satisfy the condition that $(\nexists e \in \mathtt{IE}_{L,R} \mid e \prec_{L,R} f)$. Subsequently, we construct the conflict sets for the edges in $\mathtt{IE}_{L,R}$. For the scenario described in the last paragraph, edge 5 dominates edge 6. Hence, the updated set $\mathtt{IE}_{L,R}$ is $\{7,5\}$.

5.3 Initial(α)-instrumentation

There are other opportunities to reduce the creation of conflict sets. One particularly important one entails finding specific edges in the program that are *guaranteed* to be a part of an optimal solution. These edges are identified on the basis that none of the other instrumented edges can help to infer the execution of these edges. These edges are added to $\mathcal{I}$ before applying Algorithm 3.

The edges identified by this strategy satisfy Property 3, and help to reduce the number of conflict sets generated. For example, if α is the set of edges identified by this strategy, the conflict sets containing at least one edge in α do not contribute towards obtaining the minimum hitting set, since they already satisfy the hitting set property. Hence, the final set $\mathcal{I}$ is not affected by the absence of these conflict sets. Thus, conflict sets which contain at least one edge from α need not be considered in Algorithm 3.

Since the number of pairs of paths containing an edge is exponential to the number of branch nodes present, a single edge in α will help in significantly reducing the number of useful conflict sets. Hence, the strategy to identify the set α is critical.

However, this must be achieved without affecting optimality, because incorrectly identifying an edge as part of this process can lead to a sub-optimal solution. Each edge added to α must guarantee that the cardinality of $\mathcal{I}$ using this optimization is equal to the cardinality of $\mathcal{I}$ obtained by the optimal approach.

We use the intra-procedural control-flow to identify these edges. At a branch η, if the control-flow behavior on the true edge of η is *similar* to the control-flow behavior on the false edge of η, a branch edge is *required* to be in $\mathcal{I}$. For a pair of intra-procedural straight-line paths L and R starting from either sides of the branch η, till the join node of η, let $\texttt{CS}_\texttt{L}$ and $\texttt{CS}_\texttt{R}$ represent the sequence of callee functions present in L and R respectively. If $\texttt{CS}_\texttt{L} = \texttt{CS}_\texttt{R}$, then a branch edge of η is added to α as the instrumentations in the callee cannot infer the branch edges of η. For nested conditionals, we process starting from the inner-most branch nodes and consider only the pair of paths which do not contain an edge in α.

Fig 5.1 illustrates α-instrumentation. In the four CFGs, $\texttt{L} = (1, 3)$ and $\texttt{R} = (2, 4)$. In Fig 5.1(a), since no function calls are present between the branch and join nodes $\texttt{CS}_\texttt{L} = (), \texttt{CS}_\texttt{R} = ()$. Hence, the execution of edges in this region cannot be inferred from any edge outside the region. Hence, any optimal solution will contain at least one of the four edges. We proactively include one of the edges (e.g., the marked edge 1) as part of α. Similarly, in Fig 5.1(b), even though call sites to function F are present, they are present on either side of the branch η making the behavior similar on both paths i.e. $\texttt{CS}_\texttt{L} = (\texttt{F}), \texttt{CS}_\texttt{R} = (\texttt{F})$. Even in this case, instrumentation on either edge 1 or 2 becomes necessary and one of the edges is added to α.

In contrast, Fig 5.1(c) and Fig 5.1(d) exhibit behavior that is not similar across the branches. Fig 5.1(c) contains call sites to different functions F and G, where $\texttt{CS}_\texttt{L} = (\texttt{F}), \texttt{CS}_\texttt{R} = (\texttt{G})$. Instrumented edges in F or G can help infer the execution of edges 1 and 2 which is known only after applying Algorithm 3. Similar reasoning can be used for Fig 5.1(d), where $\texttt{CS}_\texttt{L} = (\texttt{F}), \texttt{CS}_\texttt{R} = ()$. We do not add edges to α in such cases.

After identifying α, $\texttt{IE}_\texttt{L,R}$ can be further reduced. Let e be an edge in $\texttt{IE}_\texttt{L,R}$ and

let C_e be the conflict set associated with e. If any edge in α is present in C_e, then C_e satisfies Property 3 and can be ignored. Thus, e is removed from $IE_{L,R}$. Removing these edges from $IE_{L,R}$ will reduce the total number of derived conflict sets.

The two optimizations described above are used on top of Algorithm 3. As our experimental results confirm, these optimizations are effective in significantly reducing the total number of conflict sets generated. Since the dominance relation satisfies Property 2 and the α-instrumentation strategy satisfies Property 3, the correctness and the minimality of Algorithm 3 remain unaffected.

Chapter 6

Modular Design

The approach described so far provides a correct and minimal technique to derive the instrumentation but is dependent on path enumeration to determine intersecting edges and construct conflict sets.

One way to enumerate paths is to create a single graph representing the entire program, on which conflict sets are collected. This graph can be obtained by inlining the CFG of each function at its call site. For an acyclic call graph, this process results in the generation of a single whole program CFG. The size of such a graph explodes with the program size. This can be seen in Table 6.1, which shows the size of the whole program CFG for various programs from the DaCapo benchmark suite [5], where cycles in call graph are ignored. The number of nodes in the graph is large, ranging from 47K in `h2` to 2.3G in `antlr`. Consequently, path enumeration on such graphs is not practical. The table also shows the average and maximum size of the per function CFG. The average number of nodes ranges from 30 in `avrora` to 102 in `antlr`, which are orders of magnitude smaller than that of the whole program CFG. It therefore is essential that path enumeration be modular, considering paths for each function separately, and using summaries to build inter-procedural path segments.

Benchmark	DaCapo version	Count of nodes/edges in the graph					
		Inlined whole program CFG		Per function CFG			
				Avg		Max	
		Nodes (in K)	Edges (in K)	Nodes	Edges	Nodes	Edges
avrora	9.12 -bach	433	479	30	31	1316	1420
batik		69	76	34	35	1142	1222
fop		276	307	55	59	4221	4717
h2		47	53	34	36	663	716
luindex		23240	25722	39	41	724	786
lusearch		39757	44535	36	38	488	553
pmd		196	204	31	32	1126	1296
sunflow		1644	1796	41	43	1604	1603
antlr	2006-10 -MR2	2326674	2591880	102	111	1805	1970
bloat		4093	4645	50	53	1253	1368
chart		727710	822040	50	53	2161	2162
hsqldb		292	304	41	42	554	590
xalan		47	49	37	38	439	480

Table 6.1: Whole program CFG *vs.* per function CFG.

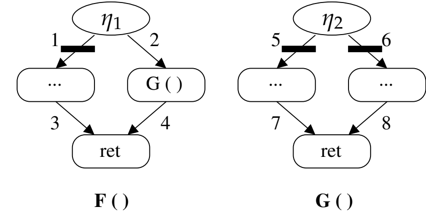


Figure 6.1: δ-instrumentation strategy.

6.1 Approximate(δ)-instrumentation

Consider a pair of paths L and R starting at a branch η in function F. Here, edges in $\text{IE}_{\text{L,R}}$ can be present beyond exit_F depending on the context in which F is called. Since context information is unavailable during the analysis of F, these intersecting edges need to be *handled* to avoid compromising correctness. We address this by instrumenting a few edges within F (by adding them to $\mathcal{I}$) before applying our analysis.

One strategy is to add the exit edge e of F to the set $\mathcal{I}$. Since any intersecting edge $\text{f} \in \text{IE}_{\text{L,R}}$ present beyond the exit_F has to go through the edge e, the conflict set created for f, C_f, will contain e. Since the edge e is already a part of $\mathcal{I}$, the conflict set C_f is resolved and can be ignored. This enables us to focus only on the edges in $\text{IE}_{\text{L,R}}$ which do not cross exit_F. However, the edge e may not be a part of the optimal instrumentation. This approximation is necessitated to ensure scalability of the analysis.

We also observe that the exit edge for a function need not be instrumented *always*. We define δ to be the set of edges instrumented using the following proposal – for each function F, if there is a path L from a branch node to the exit node of F which does not contain a call site, any edge e in L is added to δ. Otherwise, if L contains a call site

Algorithm 4 CreateSummary

σ : function $\to$ (function $\to \mathcal{P}^{\text{path}}$)
1: **procedure** CREATESUMMARY(F, σ, $\mathcal{I}$)
2: σ[F][F] := {}
3: **for** call site u in F **do**
4: **for each** path A from $\texttt{entry}_\texttt{F}$ to u **do**
5: $\texttt{E}_\texttt{A}$:= edges(_,A)
6: **if** $\texttt{E}_\texttt{A} \cap \mathcal{I} = \emptyset$ **then**
7: G := callee function of u
8: **for each** entry (H, B) in σ[G] **do**
/* *Edges to reach H from* $\texttt{entry}_\texttt{F}$ *via G* */
9: $\texttt{E}_\texttt{AB}$:= $\texttt{E}_\texttt{A} \cup$ B
10: Add $\texttt{E}_\texttt{AB}$ to σ[F][H]

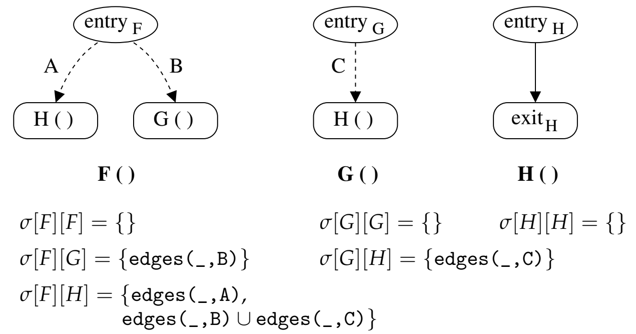


Figure 6.2: Summary generation.

to a function G, the analysis of G previously guarantees that the inter-procedural path given by L contains an edge in $\mathcal{I}$). This satisfies the condition that any path reaching the exit node of F contains an instrumented edge. Edges in δ are then added to $\mathcal{I}$.

Fig 6.1 illustrates the idea. Here, function G contains two paths from η_2 to ret – $(5,7)$ and $(6,8)$. Since both the paths do not contain a call site, one edge from each path is added to δ resulting in δ being {5,6}. Function F contains a path $(1,3)$ from η_1 to ret. Since this path does not contain any call site, we add 1 to δ. Adding edge 2 to δ is not required since all the paths originating at η_2 via 2 traverses the call site to G in which all paths are instrumented.

6.2 Function summaries

Function summaries are defined in terms of a data structure: σ : function $\to$ (function $\to \mathcal{P}^{\text{path}}$), where path $\subset \mathcal{P}^{\text{edges}}$. The summary of F ($\sigma$[F]) is a map that describes the paths to various functions reachable from F. We use edges(_,X) to represent the set of all edges in a path X. If there exists a path X from $\texttt{entry}_\texttt{F}$ to a call site of function G, this path is recorded in the summary as σ[F][G] $=$ {edges(_,X)}. Since there can be multiple such paths, σ[F] is a map from a function to multiple sets of edges. Also, as the paths present in σ are used to derive conflict sets, we require that none of the summary edges is present in $\mathcal{I}$. Otherwise, the resultant conflict

set constructed using such a summary entry will automatically satisfy the hitting set property and can be ignored.

Algorithm 4 describes the summary generation for a function F with the assumption that the summaries for the functions called by F are previously computed. The initialization on line 2 indicates that F is reachable from itself without taking any additional edge. Then, for each call site u to function G in F, we update $\sigma[\texttt{F}]$ using the entries in $\sigma[\texttt{G}]$ because the functions reachable from $\texttt{entry}_\texttt{G}$ are also reachable from $\texttt{entry}_\texttt{F}$, by traversing the path till u (e.g., A). Thus, for a function and path pair $(\texttt{H},\texttt{B})$ in $\sigma[\texttt{G}]$, $\texttt{E}_{\texttt{AB}}$ represents the sets of edges in the paths A and B, as shown in line 9. Then, $\texttt{E}_{\texttt{AB}}$ is added to $\sigma[\texttt{F}][\texttt{H}]$ at line 10.

Fig 6.2 illustrates the working of Algorithm 4. It shows the summaries generated for three functions. Since there are no call sites in H, $\sigma[\texttt{H}]$ contains a single entry. In function G, there is a path C to the call site of H. Hence, the set of edges in C is added to $\sigma[\texttt{G}][\texttt{H}]$. Function F contains call sites to G and H. Hence, $\sigma[\texttt{F}]$ is updated using $\sigma[\texttt{G}]$ and $\sigma[\texttt{H}]$. Since path B reaches the call site to G, each entry in $\sigma[\texttt{G}]$ is merged with the set of edges in B. By doing this, we get $\sigma[\texttt{F}][\texttt{G}] = \{\texttt{edges(_,B)}\}$ and $\sigma[\texttt{F}][\texttt{H}] = \{\texttt{edges(_,B)} \cup \texttt{edges(_,C)}\}$, which indicates that H is reachable from $\texttt{entry}_\texttt{F}$ by taking the edges in $\sigma[\texttt{F}][\texttt{H}]$. Similarly, we update $\sigma[\texttt{F}]$ for the call site to H and obtain the resulting σ as shown in the figure.

6.3 Conflict set generation

Because function summaries encode various control-flow paths that do not have a representative in the instrumented set of edges, our goal is to construct conflict sets to differentiate these paths. For a branch node η, we only consider the pair of intra-procedural paths L and R starting on the either side of η, such that the paths do not contain any edges in $\mathcal{I}$, and terminate either at unique call sites or at $\texttt{exit}_\texttt{F}$. If a path contains multiple call sites, we only consider the path till the first call site. Thus, L and R are intra-procedural paths. We use $\texttt{func_pairs}(\eta)$ to represent all such pairs.

Consider conflict set generation for the scenario in which L and R end at a call site. The SUMMARIZEDCONFLICTSETS procedure in Algorithm 5 explains this process. It takes as input σ apart from L and R, and updates $\mathbb{C}$ with the conflict sets associated with L and R. We define sink(X) to return the callee function associated with the call site where the path ends. Let G and H be the callee functions of the call sites in L and R respectively (line 23).

Using the summaries of the functions G and H, we create conflict sets for the entries common in $\sigma[\mathtt{G}]$ and $\sigma[\mathtt{H}]$. In other words, if there exists a function M reachable from G and H by traversing the paths in $\sigma[\mathtt{G}][\mathtt{M}]$ and $\sigma[\mathtt{H}][\mathtt{M}]$ then M is reachable from η as well. If X is an entry in $\sigma[\mathtt{G}][\mathtt{M}]$, then function M is reachable from η by taking the edges in $(\mathtt{edges(_,L)} \cup \mathtt{X})$. Similarly, we can derive a path to M via R. As there are two paths from η reaching a common function, a conflict set becomes necessary to distinguish this pair. Such a conflict set will contain the edges in $(\mathtt{edges(_,L)} \cup \mathtt{X} \cup \mathtt{edges(_,R)} \cup \mathtt{Y})$.

In the presence of multiple entries in $\sigma[\mathtt{G}][\mathtt{M}]$, let A denote the sets of edges in the paths reaching M through L. Here, A is the cross product of edges(_,L) with $\sigma[\mathtt{G}][\mathtt{M}]$ (line 25). Similarly, we use B to denote the sets of edges in the paths reaching M through R (line 26). Subsequently, we construct a conflict set for each combination in $(\mathtt{A},\mathtt{B})$ to obtain the necessary conflict sets (line 27).

The aforementioned process is to construct conflict sets when two paths under consideration have call sites at the end. Our approach still needs to be applicable in other scenarios (e.g., no call sites on pair of paths, etc). We explain our approach to handle these scenarios in the second half of GENCONFLICTSETS. We compute $\mathtt{IE}_{L,R}$ for L and R by traversing the paths (recall that L and R are intra-procedural). If L and R end at call sites, $\perp$ is not added to $\mathtt{IE}_{L,R}$. Then, we construct the conflict set for each edge in $\mathtt{IE}_{L,R}$ (lines 19 - 21).

Algorithm 5 INSTRUMENT

Input: Program $\mathcal{P}$
Output: Set $\mathcal{I}$, instrumented program $\mathcal{P}^{\mathcal{I}}$

```
1: procedure MAIN(P)
2:     I := ∅, ℂ := ∅, σ := ∅
3:     funcs := reverse topological order of call graph
4:     for function F in funcs do
5:         α := α-instrumentation on F
6:         δ := δ-instrumentation on F
7:         I := I ∪ α ∪ δ
8:         for branch η in F do
9:             GENCONFLICTSETS(η, ℂ, σ)
10:        CREATESUMMARY(F, σ, I)
11:    H := Minimum hitting set of ℂ
12:    I := I ∪ H
13:    P^I := P instrumented with edges in I

14: procedure GENCONFLICTSETS(η, ℂ, σ)
15:     for intra-procedural paths (L, R) in func_pairs(η) do
16:         if L and R end at unique call sites then
17:             SUMMARIZEDCONFLICTSETS(L, R, ℂ, σ)
18:         Compute IE_{L,R}
19:         for edge e in IE_{L,R} do
20:             C := edges(e,L) ∪ edges(e,R)
21:             Add C to ℂ

22: procedure SUMMARIZEDCONFLICTSETS(L,R,ℂ,σ)
23:     G := sink(L), H := sink(R)
24:     for function M s.t. σ[G][M] and σ[H][M] exists do
            /* Sets of edges in paths from L to M via G */
25:         A := edges(_,L) × σ[G][M]
            /* Sets of edges in paths from R to M via H */
26:         B := edges(_,R) × σ[H][M]
            /* Conflict set for each pair in (A,B) */
27:         ℂ := ℂ ∪ (A × B)
```

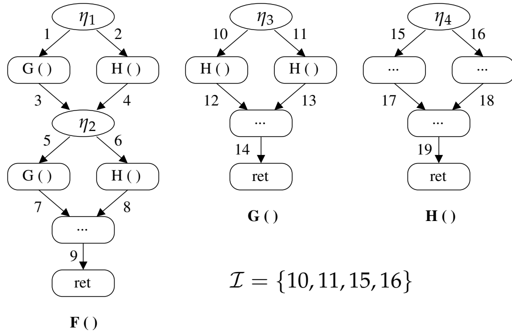
6.4 Instrumentation algorithm

Algorithm 5 summarizes our overall design to obtain $\mathcal{I}$. We perform a function-level analysis for each function F in the reverse topological order of the call graph. We initialize $\mathcal{I}$ with α and δ instrumentations, as described in previous sections (lines 5-6). Then, conflict sets are created for each η in F (lines 8-9). After the analysis of F, we create the summary (as explained in Algorithm 4) at line 10 which can be used when analyzing the callers of F. Subsequently, we apply the minimum hitting set solver that uses a greedy algorithm [11] to determine the edges for instrumentation. $\mathcal{P}^{\mathcal{I}}$ is the program obtained by instrumenting $\mathcal{P}$ with the edges in $\mathcal{I}$.

When $\mathcal{P}^{\mathcal{I}}$ is executed, a partial trace $\mathcal{L}$ is generated. We use $\mathcal{L}$ and $\mathcal{P}^{\mathcal{I}}$ as inputs to Algorithm 2 to derive the executed whole program path.

6.5 Illustrative example

We illustrate the end-end process of our approach using the program given in Fig 6.3a. The program contains three functions with multiple paths in each function. Fig 6.3b



(a) Control-flow graphs

F	η	α	δ	`func_pairs`(η)		C	σ
				L	R		
H ( )	η_4	$\{15\}$	$\{16\}$	-	-	-	H $\rightarrow$ {}
G ( )	η_3	$\{10\}$	-	-	-	-	G $\rightarrow$ {} H $\rightarrow$ {11}
F ( )	η_2	-	-	(5)	(6)	$\{5, 11, 6\}$	F $\rightarrow$ {} G $\rightarrow$ {1} H $\rightarrow$ {2}, {1, 11}
	η_1	-	-	(1)	(2)	$\{1, 11, 2\}$	

(b) Deriving instrumentation points on the illustrative example (Algorithm 5 on Fig 6.3a).

$\mathcal{L}$	Whole program path
$(10, 15, 16)$	$(1, \mathbf{10}, \mathbf{15}, 17, 19, 12, 14, 3, 6, \mathbf{16}, 18, 19, 8, 9)$
$(15, 15)$	$(2, \mathbf{15}, 17, 19, 4, 6, \mathbf{15}, 17, 19, 8, 9)$
$(11, 16, 10, 16)$	$(1, \mathbf{11}, \mathbf{16}, 18, 19, 13, 14, 3, 5, \mathbf{10}, \mathbf{16}, 18, 19, 12, 14, 7, 9)$

(c) Reconstructing WPP using the log (Algorithm 2 on Fig 6.3a).

Figure 6.3: Illustrative example.

shows the information associated with each function. To identify the instrumentation points, Algorithm 5 analyzes functions in the reverse topological order of the call graph. Hence, functions are analyzed in the order `H`, `G` and `F`.

In function `H`, since there are no call sites on either sides of η_4 satisfying our α-instrumentation criterion, we add edge 15 to α. Then, we observe that the path $(16, 18, 19)$ reaches $\texttt{exit}_{\texttt{H}}$ without making any function calls. Hence, we add edge 16 to δ. Subsequently, since `func_pairs`(η_4) is empty because the pair of paths starting at η_4 contain edges in $\mathcal{I}$ (because α and δ are added to $\mathcal{I}$), we do not construct any conflict set. Finally, the function summary $\sigma[\texttt{H}]$ contains a unique entry as there are no call sites in `H`.

Function `G` contains two call sites to `H` starting from η_3 and satisfies the α-instrumentation criterion. Hence, edge 10 is added to α. Then, as the path via edge 11 contains a call site, no edges are added to δ. The pair of paths (10) and (11) is not returned by `func_pairs`(η_3) due to the presence of edge 10 in α (therefore in $\mathcal{I}$). Hence, no conflict sets are constructed. To create the summary of `G`, we look at path (11) which reaches the function `H`. Here, $\sigma[\texttt{G}]$ is updated using $\sigma[\texttt{H}]$. Each entry in $\sigma[\texttt{H}]$ is added to $\sigma[\texttt{G}]$ along with the edge 10. By doing so, we get $\sigma[\texttt{G}][\texttt{H}] = \{11\}$, which

indicates that function `H` is reachable from `G` by taking edge 11. We do not consider the path (10) for summary generation since the edge 10 is in α.

For function `F`, none of the edges is added to α or δ as the necessary conditions are not satisfied. At branch η_2, consider the pair of paths (5) and (6) to the call sites of functions `G` and `H` respectively. To create conflict sets, we look for common entries in $\sigma[\mathtt{G}]$ and $\sigma[\mathtt{H}]$. Here, $\sigma[\mathtt{G}][\mathtt{H}] = \{11\}$ and $\sigma[\mathtt{H}][\mathtt{H}] = \{\}$ exist, which indicates that function `H` is reachable from the paths $(5, 11)$ and (6). Hence, a conflict set is created containing these edges as $\{5, 11, 6\}$. Since (5) and (6) is the only pair of path in $\mathtt{func_pairs}(\eta_2)$, this concludes the analysis of η_2. A similar analysis on the branch η_1 yields the conflict set $\{1, 11, 2\}$. Then, a function summary is created for `F` containing entries for three functions.

We observe that even though there are 36 WPPs in the program, our approach creates only two conflict sets. The minimum hitting set for the two conflict sets is $\{11\}$. We use this solution along with α and δ instrumentations to obtain the final $\mathcal{I} = \{10, 11, 15, 16\}$ which will be sufficient to distinguish the 36 WPPs.

We now explain the offline process of deriving the WPP from a partial trace $\mathcal{L}$. Fig 6.3c shows the reconstructed WPPs for three partial logs. Consider the first partial log $(10, 15, 16)$ as input to our reconstruction algorithm. Tracing starts from $\mathtt{entry_F}$ (here η_1) and the first entry in the log 10 is used to identify the next edge that needs to be traversed. We observe that the instrumented edge set $\{10, 11\}$ is reachable via 1 (before any other instrumented edge in the path) and $\{15, 16\}$ is reachable via 2 for similar reasons. This indicates that the current log entry 10 was generated by a path starting from the edge 1. Hence, we append edges 1 and 10 to the `trace` and read 15 from $\mathcal{L}$. The new entry corresponds to the branch edge at η_4. The remaining edges 17, 19, 12, 14 and 3 are appended to the `trace` until η_2 is encountered with 16 as the log entry under consideration. We derive that 6 and 16 as the reason for the entry 16 and append them along with the remaining edges up to $\mathtt{exit_F}$ to obtain the whole program path. A similar reasoning is applied to derive the WPPs from the other partial logs.

Chapter 7

Implementation

Our approach is implemented in Java on top of the Soot [35] framework to collect WPPs in Java programs. We collect the whole programs paths per thread. In this chapter, we discuss major implementation issues associated with our approach.

Callback handling A callback in a program occurs when a library method invokes an application method. Since we only trace the code in an application, it becomes challenging to identify the occurrence of each callback. To reconstruct per-thread traces, it is sufficient to track callbacks within each thread. Before the invocation of a library method, we set a thread-local variable to a unique identifier corresponding to the call site and unset it after its return. We use this identifier information at the entry of each application method to determine the occurrence of a callback.

Virtual call resolution A key requirement of our approach is to precisely identify the target method of a call site which can be problematic in the presence of virtual calls. Statically performing virtual call resolution may be imprecise and consequently can reduce the effectiveness of our approach. Therefore, we obtain the set of classes that the receiver of any virtual call site may point to at runtime using the pointer analysis built in Soot. We use this information to construct the possible target methods for each virtual call site and replace the latter with pointers to the target methods.

Reflection handling In the presence of Java reflections [30], static whole program analyses (pointer-analysis, call-graph construction, etc.) will be unsound due to lack of information about the reflective calls executed at runtime. Without sound pointer-analysis, virtual call resolution will also become unsound. To handle this, we use Tamiflex [6], which collects a reflection log for a set of recorded program runs. SOOT uses this log to fill in the missing information about the reflective calls, which aids in effectively creating sound static whole program analyses for the program.

Implicit method calls In Java, the class initialization (`clinit()`) and `finalize()` methods are called implicitly by the JVM. Control can jump to these functions at any time during execution making it hard to identify the path. We address this by explicitly instrumenting these methods to record the occurrence of the implicit method call.

Exceptions An exception breaks the normal control-flow causing the control to jump to any method in the call chain. Therefore, it is essential to record additional information about the occurrence of an exception. We handle caught exceptions by instrumenting catch blocks. Since the catch block has access to the exception object created at runtime, the source of the exception can be inferred by analyzing the stack trace. This information is recorded and then tracing proceeds normally from the catch block. Our approach handles uncaught exceptions by creating an *UncaughtExceptionHandler* for each thread and registering it in the JVM, which is invoked in the event of a crash. The handler records the stack frame entries which can be used to reconstruct the path till the crash point.

Handling recursion Algorithm 5 requires reverse topological order of the call graph necessitating the presence of an acyclic call graph. We instrument the back edges in the call graph by recording the function invocation and removing the back edge. This recording is used to infer the execution of the corresponding edge during reconstruction.

Chapter 8

Experimental Evaluation

We conducted our experiments on an Ubuntu-14.04 desktop machine having a 4.0GHz Intel Core i7 processor and 32GB of RAM, running Java HotSpot(TM) 64-Bit Server VM (version 1.8). We evaluated our approach using the DaCapo benchmark suite [5]. The programs from the 2009 version as well as the 2006 version of DaCapo were used. A few programs in the 2009 version were not supported by Soot [8; 9; 33]. The details of the programs analyzed and the versions used are given in Table 6.1. We ran the experiment 100 times with 'default' input and report the average of these runs.

We compare our approach (WPP-O) with three other techniques (WPP-M, WPP-L and WPP-B). WPP-M corresponds to the baseline for instrumentation, WPP-L is the approach proposed by Larus [19] and WPP-B is an approach that instruments the outgoing edges of all branch nodes. WPP-M is obtained by modifying Algorithm 5, where δ-instrumentation is the only source of approximation[1] as edges in δ need not be present in the optimal solution $\mathcal{I}$. Since $\mathcal{I} - \delta$ is derived using optimal techniques, *any* optimal solution for the overall program will instrument at least as many edges as in $\mathcal{I} - \delta$. While instrumenting $\mathcal{I} - \delta$ alone is insufficient to unambiguously reconstruct any path in the program, it serves as a baseline to perform our comparisons.

[1]The approximations to scale the approach for real programs.

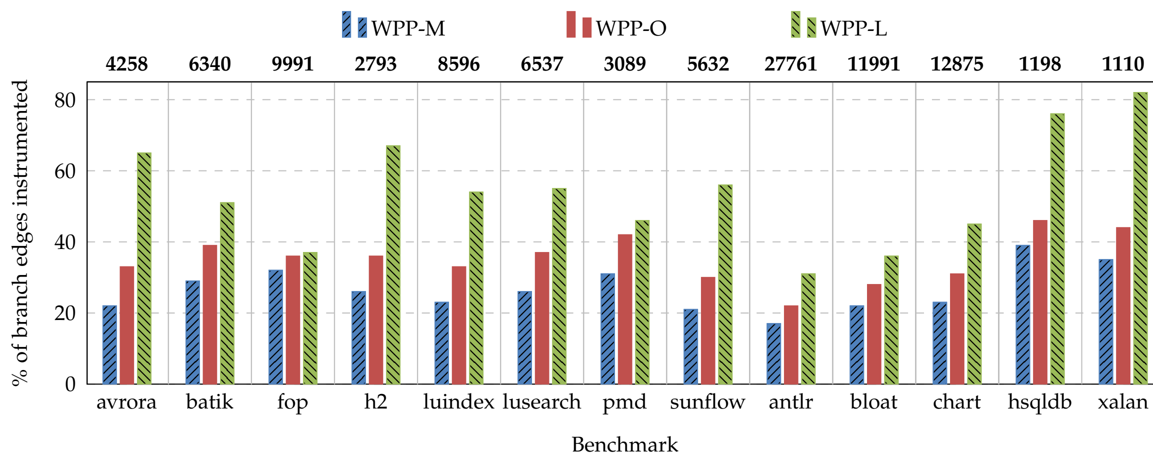
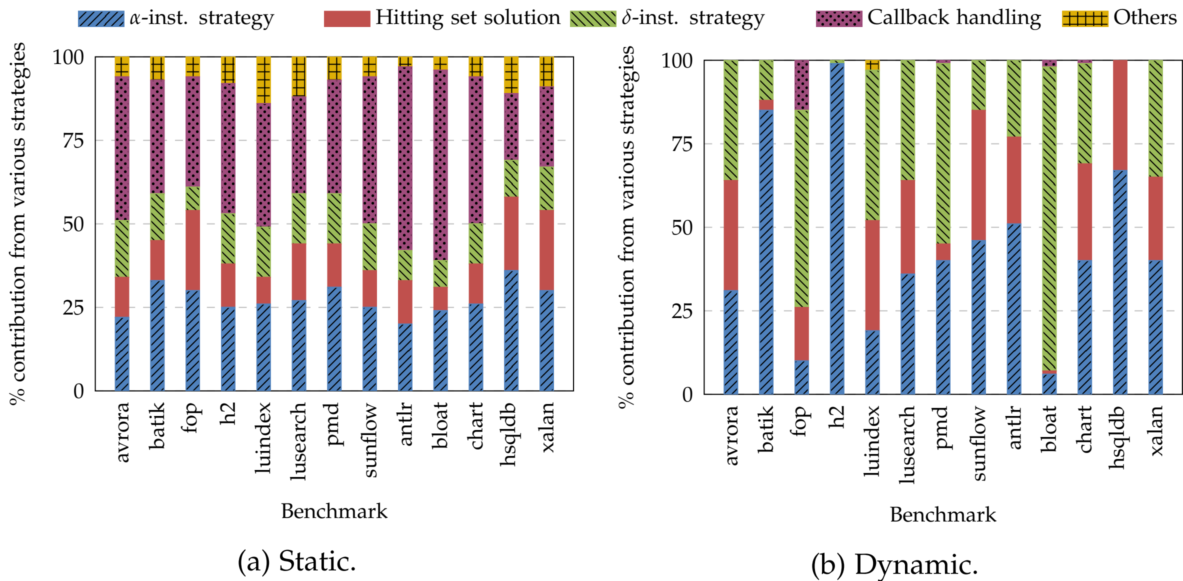


Figure 8.1: Static instrumentation comparison.

Our approach instruments fewer edges Fig 8.1 shows the number of edges instrumented by various approaches as a percentage of total branch edges present in the program. The numbers on top of each benchmark show the total number of branch edges. The minimum number of edges that *need* to be instrumented is 27% on average. In comparison, on average, WPP-O and WPP-L instrument 35% and 56% of the branch edges respectively. The difference in this number is because WPP-O skips the instrumentation of certain branch edges which can be inferred by other edges. This reduction in instrumentation can consequently reduce the incurred runtime overhead.

Characterization of the instrumentation introduced by our approach Fig 8.2a characterizes the static instrumentation introduced by various components of WPP-O. The instrumentations inserted for recursion handling, implicit method handling, and exception handling are grouped together as `Others`. Broadly, the edges identified by α-instrumentation (28% on average) and by the hitting set solution (15% on average) are necessary for precise reconstruction. δ-instrumentation edges (13% on average) introduce the approximation for scaling the approach to real programs. Callback handling (38% on average) and other instrumentation (6% on average) account for handling Java-specific features. The Java-specific instrumentation is a necessary component to collect WPPs for Java programs by any approach. The approximations



(a) Static.

(b) Dynamic.

Figure 8.2: Classification of instrumented edges.

associated with the hitting set solution and δ-instrumentation are potential sources of additional instrumentation which is capped to 28% on average.

In comparison, Fig 8.2b characterizes the dynamic instrumentations recorded by various components of WPP-O. A significant portion of the instrumentations is due to the edges due to α-instrumentation, hitting set solution and δ-instrumentation. The instrumentations due to callback handling and other Java-specific instrumentations are non-existent, except for `fop`, where the total number of instrumentations recorded is less. This demonstrates the importance of the approach proposed in this thesis to reduce the instrumentation based on program structure.

The time overhead associated with our approach is less Fig 8.3 shows the time overhead comparisons across the various approaches, where the y-axis represents the percentage overhead in log scale. The average runtime overhead incurred by the approach that provides the baseline (WPP-M) is 71% and the overhead by WPP-O is 97%. This indicates that the runtime overhead incurred by an optimal, but technically intractable approach lies between 71% and 97%. In contrast, the average overhead incurred by WPP-L is 278%. Further, WPP-O also shows an average and maximum

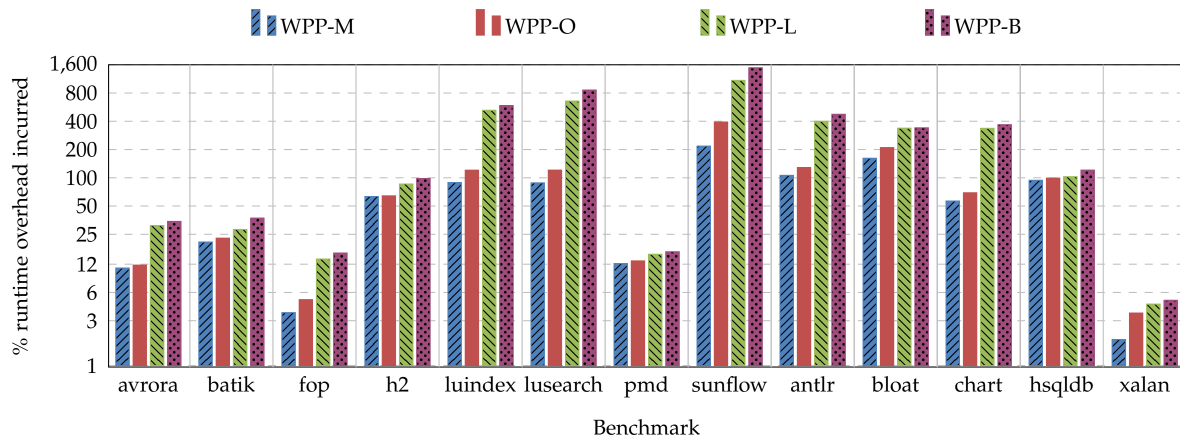


Figure 8.3: Time overhead comparison in log scale.

performance improvement of 2.57x and 5.42x respectively, as compared to WPP-L.

Static instrumentation *vs* Runtime overhead The runtime overhead may not correlate with the number of edges instrumented statically. For example, in `fop`, even though the improvement in static instrumentation count between WPP-O and WPP-L is not large, we see a significant improvement at runtime. On the contrary, in `xalan`, the improvement in static instrumentation count is substantially higher than that of the runtime overhead. This behavior can be accounted due to the design of WPP-O where all branch edges are considered equally likely and the dynamic branch behavior of the program is not considered. Consequently, instrumenting a *hot* edge will result in a higher overhead. Leveraging the profile of executed edges to determine the minimum instrumentation count (even at runtime) is part of future work.

Characterization of the trace size Fig 8.4 shows the size of the trace generated by various approaches as a percentage of the trace size generated by instrumenting all branch edges. The labels on top show the actual trace size by instrumenting all branch edges. On average, WPP-M, WPP-O and WPP-L show a reduction in trace size to 38%, 56% and 72% respectively. Interestingly, in `batik`, `h2` and `hsqldb`, the size of the trace generated by WPP-L is less than that of WPP-O, even though the runtime overhead is much higher. This is because the Ball and Larus [3] numbering scheme

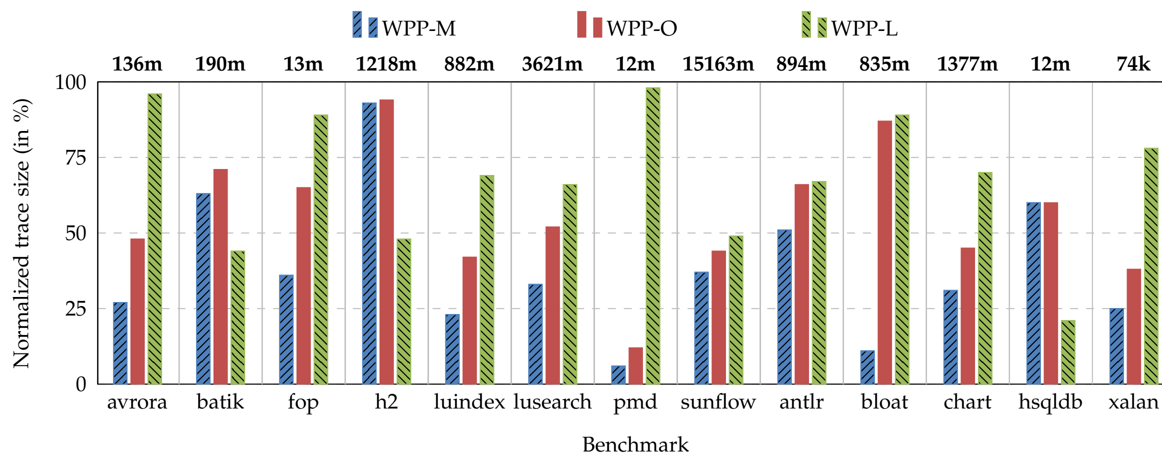


Figure 8.4: Reduction in trace size.

is employed by WPP-L to encode the generated instrumentations succinctly. This helps in reducing the size of the trace because each log entry can encode a large path component as compared to WPP-O. Incorporating a numbering scheme on top of the approach proposed in this thesis to reduce trace size is part of future work.

Impact of α-instrumentation Fig 8.5 demonstrates the effectiveness of the α instrumentation strategy in reducing the conflict sets collected. The graph shows the number of conflict sets as a percentage of the number of conflict sets collected without using the α-instrumentation strategy. The labels on top of bars show the actual number. We observe that the number of conflict sets is not more than 9%. This is because the total number of conflict sets collected grows exponentially with the program size, and proactively selecting edges to instrument will help avoid the creation of certain conflict sets. Since the creation of such conflict sets is avoided on the basis of Property 3, the reduced number of the conflict sets does not compromise on optimality.

Analysis time Table 8.1 shows the time taken by the various phases of our approach. The pre-analysis stage involves employing Algorithm 5 to identify the set of instrumentation points and generating the instrumented program. The average time to identify the instrumented edges is 26 seconds. The post-analysis stage regenerates the whole program path executed, by employing Algorithm 2 using the generated

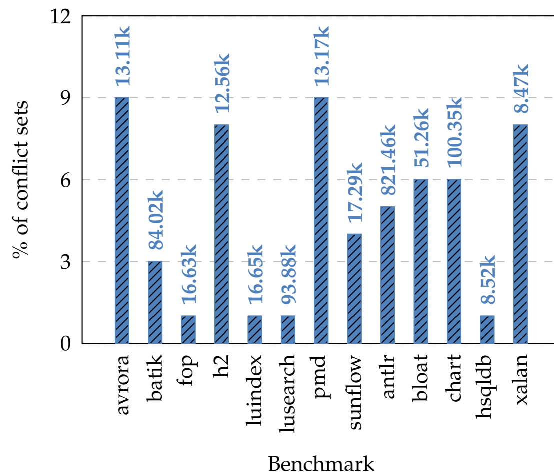


Figure 8.5: Impact of α-instrumentation strategy.

Benchmark	Analysis time (sec)	Runtime (sec)	Regeneration time (sec)	Total (sec)
avrora	10	1.94	13	24.94
batik	18	2.2	21	41.20
fop	36	1.09	37	74.09
h2	12	7.85	23	42.85
luindex	17	1.6	21	39.60
lusearch	14	2.21	20	36.21
pmd	14	1.55	16	31.55
sunflow	16	8.35	31	55.35
antlr	122	1.92	128	251.92
bloat	28	4.6	37	69.60
chart	22	3.72	29	54.72
hsqldb	8	3.62	12	23.62
xalan	19	1.46	19	39.46

Table 8.1: Time comparison.

log. The average time to obtain WPP from the logged partial trace is 31 seconds. The negligible time spent in offline static analyses helps in efficient instrumentation, resulting in reduced runtime overheads while generating precise WPPs.

Chapter 9

Related Work

Path tracing The problem of finding a minimum number of program points to profile or trace a program has been previously reduced to existing graph problems [18; 28]. Then, the tracing problem is proven to be NP-complete for single procedure programs [23]. However, these approaches cannot be directly extended to programs with multiple procedures. To reduce the overall instrumentation in these programs, Larus [19] proposed an efficient solution using the Ball and Larus [3] numbering scheme.

Path profiling The seminal work on path profiling by Ball and Larus [3] is used extensively to identify the frequency of paths executed within each function. Further, a number of approaches have been built on this idea [36; 13]. Vaswani et al. [36] focuses on profiling a subset of the paths present in a function, and D'Elia and Demetrescu [13] extends the profiling algorithm to identify the hot paths spanning multiple loop iterations. In our work, since the WPP consisting of the sequence of executed edges can be derived, an offline analysis can be employed to extract the path profiles, edge profiles, etc.

Call trace collection CASPER [37] focuses on reducing the number of call sites monitored at runtime by leveraging program structure to identify call sites that can

be inferred. Even though Casper can be modified to collect path traces, there are a few shortcomings. Casper formulates the problem using a modified version of an LL(1) grammar, which does not precisely represent the minimum number of call sites needed to obtain the call trace. Also, Casper obtains the *optimal* instrumentation within each method, which is sub-optimal with respect to the whole program. Our approach formulates the problem using conflict sets and we prove the minimality of instrumentation in the context of the whole program.

Calling context encoding Calling context encoding [7; 31; 40; 38] focuses on identifying the precise calling context at each point in the program. For example, Sumner et al. [31] proposed an approach which extends the Ball and Larus [3] algorithm for the call graph. This information can further be used in various performance optimizations. The WPP can also be used to extract the calling context, albeit at a higher cost, as it collects the entire information about the execution.

Data dependence trace Data dependence traces capture the dynamic dependence between various memory locations during an execution [32; 17; 16]. This information is useful in finding opportunities for exploiting parallelism in programs. For example, Kim et al. [17] identify various loops that can be parallelized by looking at the different memory locations accessed within the loop. Our work is orthogonal to these techniques as we focus only on the control-flow of the program.

Applications WPPs have been used to identify the hot regions in a program [25; 29]. This information is useful because compiler optimizations can then be performed on this region to obtain better runtime performance. WPPs have been used to improve the reliability of software. For example, Tikir and Hollingsworth [34] used executed WPPs as a criterion to track the effectiveness of a test suite. WPPs are also helpful in identifying software thefts, where software is watermarked based on the executed paths [24]. It has been used in concurrency debugging, where concurrency defects are detected using the intra-thread whole program trace [14; 21; 22]. These applications

can all benefit from the approach proposed in this thesis.

Chapter 10

Conclusions

This thesis presents a new technique to precisely and efficiently reconstruct a program execution's control-flow, represented as a whole-program path. Its key insight is the ability to infer edge instrumentation points for the contexts in which a function call executes, solely from the instrumentation points generated within the function. This context includes both paths that lead to the call as well as paths that are executed by the function's continuation. Incorporating this insight within a modular and scalable analysis yields results close to the optimal on the applications we have studied, and significantly outperforms the state-of-the-art.

Bibliography

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 72–84, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277665. URL http://doi.acm.org/10.1145/277650.277665.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994. ISSN 0164-0925. doi: 10.1145/183432.183527. URL http://doi.acm.org/10.1145/183432.183527.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7641-8. URL http://dl.acm.org/citation.cfm?id=243846.243857.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251272.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović,

T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. URL http://doi.acm.org/10.1145/1167473.1167488.

[6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985827. URL http://doi.acm.org/10.1145/1985793.1985827.

[7] 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*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297035. URL http://doi.acm.org/10.1145/1297027.1297035.

[8] Bug-651. Github issue. https://github.com/Sable/soot/issues/651, 2016.

[9] Bug-652. Github issue. https://github.com/Sable/soot/issues/652, 2016.

[10] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based faliure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251175.1251198.

[11] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[12] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008. ISBN 0073523402, 9780073523408.

[13] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 373–390, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509521. URL `http://doi.acm.org/10.1145/2509136.2509521`.

[14] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 141–152, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462167. URL `http://doi.acm.org/10.1145/2491956.2462167`.

[15] J. C. Huang. Program instrumentation and software testing. *Computer*, 11 (4):25–32, Apr. 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218134. URL `http://dx.doi.org/10.1109/C-M.1978.218134`.

[16] A. Ketterlin and P. Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 437–448, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. doi: 10.1109/MICRO.2012.47. URL `http://dx.doi.org/10.1109/MICRO.2012.47`.

[17] M. Kim, H. Kim, and C.-K. Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 535–546. IEEE Computer Society, 2010.

[18] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313–322, 1973.

[19] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages

259–269, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301678. URL `http://doi.acm.org/10.1145/301618.301678`.

[20] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL `http://dl.acm.org/citation.cfm?id=776816.776854`.

[21] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 586–595, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737973. URL `http://doi.acm.org/10.1145/2737924.2737973`.

[22] N. Machado, B. Lucia, and L. Rodrigues. Production-guided concurrency debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 29:1–29:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4092-2. doi: 10.1145/2851141.2851149. URL `http://doi.acm.org/10.1145/2851141.2851149`.

[23] S. Maheshwari. Traversal marker placement problems are np-complete. *Boulder Univ. Report CU-CS-092*, 76, 1976.

[24] G. Myles and C. Collberg. *Detecting Software Theft via Whole Program Path Birthmarks*, pages 404–415. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-30144-8. doi: 10.1007/978-3-540-30144-8_34. URL `http://dx.doi.org/10.1007/978-3-540-30144-8_34`.

[25] A. Neustifter. *Efficient profiling in the LLVM compiler infrastructure*. na, 2010.

[26] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 378–388. IEEE, 2013.

[27] R. L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Trans. Softw. Eng.*, 8(1):34–42, Jan. 1982. ISSN 0098-5589. doi: 10.1109/TSE.1982.234772. URL http://dx.doi.org/10.1109/TSE.1982.234772.

[28] C. Ramamoorthy, K. Kim, and W. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE transactions on Software Engineering*, (4):403–411, 1975.

[29] B. Scholz and E. Mehofer. Dataflow frequency analysis based on whole program paths. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 95–103, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1620-3. URL http://dl.acm.org/citation.cfm?id=645989.674305.

[30] H. Shildt. *Java: The complete reference*. McGraw-Hill, 2007.

[31] 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*, ICSE '10, pages 525–534, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806875. URL http://doi.acm.org/10.1145/1806799.1806875.

[32] S. Tallam, R. Gupta, and X. Zhang. Extended whole program paths. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 17–26, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. doi: 10.1109/PACT.2005.22. URL http://dx.doi.org/10.1109/PACT.2005.22.

[33] TamiflexBug-4. Github issue. https://github.com/secure-software-engineering/tamiflex/issues/4, 2016.

[34] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on*

Software Testing and Analysis, ISSTA '02, pages 86–96, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. doi: 10.1145/566172.566186. URL http://doi.acm.org/10.1145/566172.566186.

[35] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 18–34, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67263-X. URL http://dl.acm.org/citation.cfm?id=647476.727758.

[36] 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*, POPL '07, pages 351–362, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190268. URL http://doi.acm.org/10.1145/1190216.1190268.

[37] R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang. Casper: An efficient approach to call trace collection. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 678–690, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837619. URL http://doi.acm.org/10.1145/2837614.2837619.

[38] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. Deltapath: Precise and scalable calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 109:109–109:119, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544150. URL http://doi.acm.org/10.1145/2544137.2544150.

[39] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 180–190, New

York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378835. URL http://doi.acm.org/10.1145/378795.378835.

[40] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134012. URL http://doi.acm.org/10.1145/1133981.1134012.