Optimizing Customized Program Coverage

Peter Ohmann
ohmann@cs.wisc.edu

David Bingham Brown
bingham@cs.wisc.edu

Naveen Neelakandan
neelakandan@cs.wisc.edu

Jeff Linderoth
linderoth@wisc.edu

Ben Liblit
liblit@cs.wisc.edu

University of Wisconsin–Madison
Madison, WI, USA

ABSTRACT

Program coverage is used across many stages of software development. While common during testing, program coverage has also found use outside the test lab, in production software. However, production software has stricter requirements on run-time overheads, and may limit possible program instrumentation. Thus, optimizing the placement of probes to gather program coverage is important.

We introduce and study the problem of customized program coverage optimization. We generalize previous work that optimizes for complete coverage instrumentation with a system that adapts optimization to customizable program coverage requirements. Specifically, our system allows a user to specify desired coverage locations and to limit legal instrumentation locations. We prove that the problem of determining optimal coverage probes is NP-hard, and we present a solution based on mixed integer linear programming. Due to the computational complexity of the problem, we also provide two practical approximation approaches. We evaluate the effectiveness of our approximations across a diverse set of benchmarks, and show that our techniques can substantially reduce instrumentation while allowing the user immense freedom in defining coverage requirements. When naïve instrumentation is dense or expensive, our optimizations succeed in lowering execution time overheads.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging; Software post-development issues;** *Software performance;* •**Mathematics of computing** → *Integer programming;*

Keywords

Debugging, mixed integer linear optimization, program coverage

1. INTRODUCTION

Program coverage data identifies which program features executed during one or more runs of a program. Program coverage is commonly used as a quality metric for test suites; developers wish to ensure that a test suite exercises an adequate portion of their code-base. However, developers also use program coverage in other contexts, such as postmortem program analysis [28, 29] and fault localization [22, 36]. Many granularities of coverage are possible. For example, statement coverage identifies the set of statements that executed during a run. Coarser granularities, such as function coverage, gather coverage data for a smaller set of program points, providing less detailed information, but with lower run-time overhead. Finer granularities, such as path coverage, make the opposite trade: higher run-time overhead for more detailed execution information. This paper targets uses of program coverage in low-overhead monitoring of deployed applications. Specifically, we optimize instrumentation for binarized control-flow coverage metrics (i.e., coverage data marks each program location as "covered" or "uncovered," rather than counting the number of occurrences). We focus on statement and edge coverage, which prior work has shown to be useful in postmortem debugging [29] and amenable to residual monitoring [32, 34]. That is, we reduce the instrumentation required to track which program points were covered for a particular execution[1].

In many scenarios, one may not require—or cannot afford to gather—full coverage information for a program run. This is especially true after deployment: deployed software is already partially tested, and only tolerates small run-time overheads. Sparse tracing also means sending less data back to developers in field reports. Further, developers often do not require full information from these reports. For example, developers may focus on code features (e.g., call sites [28, 29]) likely to be useful for debugging or program analysis. Alternately, they may desire coverage only for newly added code, or code not adequately tested before release [32, 34]. Conversely, security-sensitive code, tightly optimized code, or code with strict real-time requirements may be off-limits for monitoring.

Optimization for gathering program coverage is well-studied (see Section 5). Agrawal [1] and Tikir and Hollingsworth [37] optimize probe placement for binarized statement coverage. Unfortunately, Agrawal's approach is not applicable for incomplete executions, such as those that terminate unexpectedly due to a fatal signal. Support for aborted runs is important if post-deployment failure data is to include coverage information [29]. However, low-cost coverage tracing is especially important in these deployed scenarios, as only small amounts of run-time overhead are tolerable. Further, neither Agrawal nor Tikir and Hollingsworth optimizes for incomplete or constrained coverage requirements. Deployed software calls for *customized* coverage information: coverage at a subset of program locations, such as those not exercised by a test suite [34] or those containing features interesting for debugging [28, 29].

In this paper we present three approaches that select a smaller set of coverage probes given instrumentation restrictions and a set of program points of interest. We prove that the resulting problem

[1]Source code is available as part of the CSI instrumentation framework at http://pages.cs.wisc.edu/~liblit/ase-2016-b/code/.

To appear in the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), September 2016, Singapore.

is NP-hard, so the existence of a fast algorithm for computing the optimal set of probes is unlikely. Our first approach constructs a mixed integer linear program (MILP) whose solution identifies the optimal probe set; however, solving this MILP to optimality requires prohibitive analysis time during instrumentation. We therefore offer two approximation approaches. The first approximation is very inexpensive to compute, but provides no optimality guarantees; the second finds a locally optimal approximation. The primary contributions of this work are as follows:

- We define the Customized Coverage Probing Problem and formalize the notion of a coverage set for customized coverage criteria. We argue the problem's significance and applicability to common problems in monitoring deployed applications.
- We prove that the Customized Coverage Probing Problem is NP-hard via a reduction from work by Maheshwari [23].
- We present three approaches to tackle this problem: an optimal solution implemented as a MILP, and two approximations with varying degrees of optimality and computation cost.
- We perform an extensive evaluation of our approaches. In agreement with prior work on full-coverage optimization [1, 27, 37], we find that even coarse approximations can statically eliminate much instrumentation. When naïve instrumentation imposes large overhead, our optimizations significantly reduce both compilation and run-time costs.

This paper is organized as follows. Section 2 formally defines the Customized Coverage Probing Problem, and argues for its applicability to common problems using program coverage for deployed applications. Section 3 describes our three approaches, which we then evaluate in Section 4. Section 5 positions our work in the context of prior research. Section 6 concludes.

2. CUSTOMIZED COVERAGE

Our goal is to determine an optimal instrumentation plan to gather *customized, binarized* program coverage information. More concretely, we are given a single procedure's control-flow graph (CFG), G; some subset of the vertices in G for which the developer desires information, D; and another subset of the vertices in G defining legal observation points, I. The problem is to determine the cheapest set of *probes* to insert into locations from I such that, for any given path p through G, the set of probes encountered along p is sufficient to determine which vertices from D were traversed along p.

2.1 Input

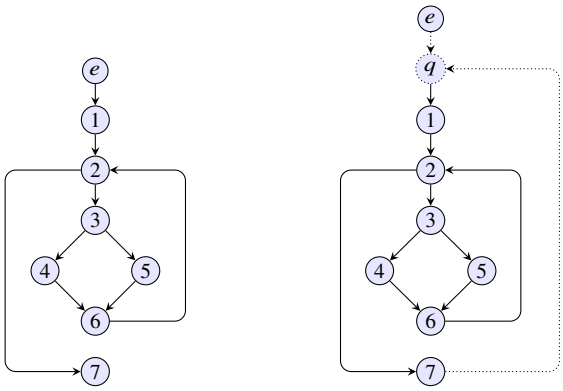
The input to the problem is as follows:

- $G = (V, E)$, a directed graph with vertices V and edges E
- $e \in V$, a unique source (or entry) vertex with in-degree 0
- $I \subseteq V$, a subset of vertices that may be probed (instrumented)
- c_i, the cost of probing vertex $i \in I$, where $\forall i \in I, c_i > 0$
- $D \subseteq V$, a set of "desired" vertices that must be "covered"
- $X \subseteq V$, a set of possible termination points

2.2 Problem Definition

Definition 1 *For $v_1, v_2 \in V$, $v_1 \rightarrow v_2$ denotes the set of all paths from v_1 to v_2 in G. If $v_1 = v_2$, then the trivial path (crossing no edges) is included in this set.*

Definition 2 *$V(p)$ denotes the set of all vertices encountered along path p, including the start and end vertices of p. If p is the trivial path from vertex v (crossing no edges), then $V(p)$ is the singleton set $\{v\}$.*



(a) Original control-flow graph **(b) Multiple function executions**

Figure 1: Example control-flow graph with transformation for multiple function executions

Definition 3 *A set of vertices $S \subseteq I$ is called a **coverage set** of D if $\forall x \in X$ and $\forall p_1, p_2 \in e \rightarrow x$, $V(p_1) \cap S = V(p_2) \cap S \implies V(p_1) \cap D = V(p_2) \cap D$. That is, two executions ending at the same termination point $x \in X$ that encounter the same S vertices must encounter the same D vertices as well. Contrapositively, paths that encounter different D vertices must be distinguishable by encountering different S vertices as well.*

Problem Statement:

The **Customized Coverage Probing Problem** is to find a coverage set S of D such that $\sum_{s \in S} c_s$ is minimal.

2.3 Discussion

We desire the lowest-cost coverage set of D. Put another way, the goal is to find the cheapest set of probe vertices ($S \subseteq I$) such that the resulting observations ($V(p_1) \cap S$ and $V(p_2) \cap S$) are sufficient to derive desired coverage information ($V(p_1) \cap D$ and $V(p_2) \cap D$) for all executions, whether terminating normally or abnormally ($\forall x \in X$). If $D \subseteq I$, then a coverage set of D exists, since we can set $S = D$. However, the existence of some S that satisfies Definition 3 does not imply that $D \subseteq I$. That is, we may be able to infer coverage information for $d \in D$ without probing d directly.

The cost of probing each vertex (c_i) is input to the problem. To minimize the expected run-time cost of instrumentation, these values should represent the expected execution frequency of each $i \in I$, which could be derived from static heuristics [38] or profile data [5, 18]. However, other cost functions can minimize the static number of probes inserted, or prioritize instrumentation locations based on real-time requirements or security concerns. (See Section 4 for computation of these values in our evaluation.)

2.4 Encoding Specific Problems

We now describe how to encode specific realistic requirements as instances of our problem. Figure 1a shows an example CFG with entry vertex e. Using this graph as $G = (V, E)$, one can obtain full *local* statement coverage data with $I = D = V \setminus \{e\}$. Assuming the program could crash at any statement, we can also let $X = V$. We thus obtain full coverage data for any local execution (i.e., if coverage information is stored in the program stack [29]), and for functions known to execute exactly once (e.g., the `main` function). To handle multiple executions of the instrumented function, one performs a simple graph modification: add a new vertex q repre-

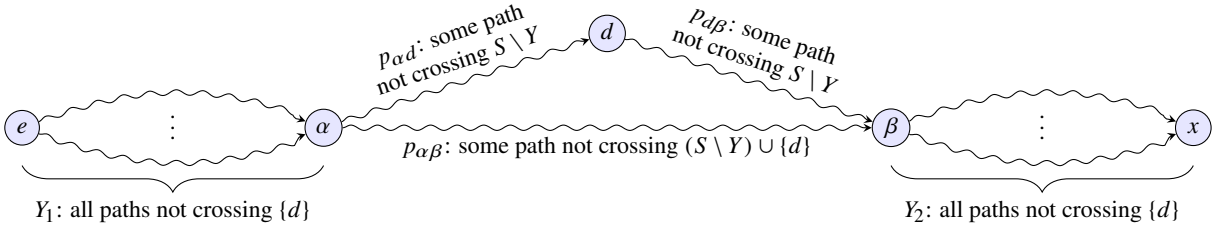


Figure 2: Pictorial representation of an ambiguous triangle

senting any execution from the rest of the program, as shown in Fig. 1b. For full statement coverage, we then let $I = D = V \setminus \{e, q\}$. Usually, we say that $q \in X$, indicating that execution may terminate outside the current function. If we can statically determine that the program cannot crash in the function (or we only need coverage data for complete executions), we can set $X = \{q\}$. For edge coverage instead of vertex coverage, we split each edge in the CFG. Let V' be the set of new vertices added on these split edges. Full edge coverage is then encoded by $I = D = V'$.

In some contexts, one need only gather coverage information for a specific set of statements or edges. For example, prior work in debugging and program slicing focuses on call statements [28, 29]. In this case, D is the set of basic blocks containing call statements. Other work gathers residual coverage information for deployed applications based on statements or branches not covered during in-house testing [32, 34]. Here, D is the set of non-covered blocks (or edges, using the edge-splitting method described above). Beyond well-studied cases, many others are possible. One might reduce runtime cost by removing some portion of the "hottest" blocks from I as identified by standard profiling. With previous post-deployment failure analysis data, one might set D to program locations whose execution status was unknown in the failing execution. After a product release, developers could isolate new failures by setting D to recently changed code. Alternately, one might exclude security-sensitive code or code with real-time requirements from I.

2.5 Proof of NP-Hardness

To show that the Customized Coverage Probing Problem is NP-hard, we show that a known NP-complete problem reduces to our problem in polynomial time. Maheshwari [23] proves that determining the optimal placement for traversal markers in acyclic CFGs is NP-complete. Specifically, given an acyclic CFG, $G = (V, E)$, one can enumerate the set of source–sink paths through G, $P(G)$; then, for all $p \in P(G)$, define $E(p)$ as the set of edges along p. Maheshwari proves that determining the minimum-size set of edge traversal markers, $M \subseteq E$, such that each $p \in P(G)$ traverses a unique set of traversal markers, is an NP-complete problem. Formally, the problem is to find $M \subseteq E$ such that for all distinct $p_1, p_2 \in P(G)$, $M \cap E(p_1) \neq M \cap E(p_2)$; and for all M' such that $|M'| < |M|$, there exist distinct $p_1, p_2 \in P(G)$ such that $M' \cap E(p_1) = M' \cap E(p_2)$.

Note that traversal markers and coverage probes yield precisely the same information for acyclic graphs: no edge may occur more than once in any path (because there are no cycles), and the order of any pair of edges is fixed if both may occur along the same path (because there would otherwise be a path containing a cycle). Thus, while Maheshwari's original intent was to prove that minimal probing to distinguish all *paths* in an acyclic CFG is NP-complete, the same proof also shows that minimizing the number of edge probes to obtain *full edge coverage* information is NP-complete.

Transforming a Traversal Marker Placement Problem into a Customized Coverage Probing Problem is straightforward. Given a CFG $G = (V, E)$, we split each edge to instrument for edge coverage as in Section 2.4. Then, an optimal solution to the Customized Coverage Probing Problem with input

$$I = D = V' \setminus V \qquad c_i = 1 \text{ for all } i \in I$$

is also an optimal traversal marker placement. The transformation is polynomial, as splitting each edge is simply an $O(E)$ operation. Therefore, the Customized Coverage Probing Problem is NP-hard.

3. APPROACH

This section describes our solution to the Customized Coverage Probing Problem, and two approximations. First, in Section 3.1, we give a characterization of a coverage set that can be checked in polynomial time. Using this new characterization, Section 3.2 outlines a MILP to identify the optimal coverage set. Section 3.3 describes a dominance-based approximation; however, this approximation provides no optimality guarantees. We build on this approach in Section 3.4 to formulate a second approximation that finds a locally minimal coverage set. Finally, Section 3.5 touches on recovery of full coverage information from optimized instrumentation.

3.1 Checking Sufficiency of Coverage Sets

As a first step, we must devise a sufficiency check for a candidate coverage set $S \subseteq I$. That is, we would like a simplified condition (relative to Definition 3) in order to check whether any given set S is a coverage set of D. Recall that D is the set of desired vertices, and that S is a coverage set of D if and only if coverage information for S unambiguously allows one to determine coverage information for D on any execution path. Definition 3 is given with respect to all paths through G ending at some $x \in X$; for any graph with loops, there may be infinitely many such paths. Intuitively, however, many of these paths are redundant with respect to S's coverage set status. We formalize this intuition by narrowing our search to finding "ambiguous triangles" in which some $d \in D$ may or may not occur between observation vertices α and β. The condition is represented pictorially in Fig. 2. As input, we have all items from Section 2.1 and some $S \subseteq I$, a candidate coverage set for D.

In Fig. 2, wavy lines represent paths crossing 0 or more edges. The core of this approach lies in finding an ambiguous triangle composed of paths $p_{\alpha d}$, $p_{\alpha\beta}$, and $p_{d\beta}$. Such a triangle represents an ambiguous region of execution (between observation points) that demonstrates that S alone is not sufficient to determine if d executed. Conversely, S is a coverage set of D if S allows no ambiguous triangles in G. An ambiguous triangle is a triple (α, β, d) such that:

- α is either the entry vertex or an observation point from S,
- β is either from S or a possible termination location, and
- $d \in D$, and d may or may not occur on paths $\alpha \to \beta$ that contain no other new observations.

To define *new observations* above, we must define sets Y_1, Y_2, and Y. The set Y_1 contains all d-free paths from the entry vertex to α. The set Y_2 contains all d-free paths from β to a possible termination location. (If $d \in X$, then d is excluded as a possible final location, since ending at d implies the observation, and hence execution, of d.) The set Y contains all vertices along paths in $Y_1 \cup Y_2$. These vertices may occur before or after $\alpha \rightarrow \beta$ paths, and, hence, any vertices in $Y \cap S$ provide no new information along $\alpha \rightarrow \beta$ paths. If $Y_1 = \emptyset$, then d dominates α on all paths through G; hence, α will not provide a useful witness, as d occurs on *all* paths through α. Conversely, if $Y_2 = \emptyset$, then d post-dominates β with respect to all termination points; hence, β will not provide a useful witness, as d will occur on *all* paths through β. Note that we can form paths p_1 and p_2 from Definition 3 by conjoining appropriate paths from Y_1 and Y_2 to the two paths through the ambiguous triangle.

To formalize the above, we require one new definition:

Definition 4 ***(Connected Excluding)*** *For $\Psi \subseteq V$ and $v_1, v_2 \in V$, let $v_1 \xrightarrow{\notin \Psi} v_2$ denote the set of paths from v_1 to v_2 that* do not *cross any edges with a source or target vertex $\psi \in \Psi$.*

This definition includes trivial paths. That is, for $v \in V$, $v \xrightarrow{\notin \Psi} v$ is nonempty for any $\Psi \subseteq V$, even if $v \in \Psi$. Then, for all (α, β, d) triples where

$$\alpha \in S \cup \{e\} \qquad \beta \in S \cup X \qquad d \in D \setminus S$$

we define the following sets:

$$Y_1 = e \xrightarrow{\notin \{d\}} \alpha \qquad P_{\alpha d} = \alpha \xrightarrow{\notin S \setminus Y} d$$

$$Y_2 = \bigcup_{x \in X \setminus \{d\}} \beta \xrightarrow{\notin \{d\}} x \qquad P_{\alpha\beta} = \alpha \xrightarrow{\notin (S \setminus Y) \cup \{d\}} \beta$$

$$Y = \bigcup_{\pi \in Y_1 \cup Y_2} V(\pi) \qquad P_{d\beta} = d \xrightarrow{\notin S \setminus Y} \beta$$

Then, set S is a coverage set of D if and only if:

$$Y_1 = \emptyset \ \vee Y_2 = \emptyset \ \vee P_{\alpha d} = \emptyset \ \vee P_{\alpha\beta} = \emptyset \ \vee P_{d\beta} = \emptyset$$

for all (α, β, d) triples defined above. Note that these five disjuncts correspond precisely to the five necessary parts of the ambiguous triangle pictured in Fig. 2, and those necessary to form paths p_1 and p_2 from Definition 3. Thus, if all five of these subpaths exist for any (α, β, d) triple, then S is not a coverage set of D.

As an example, consider the CFG in Fig. 1a, and the input configuration $X = \{6\}$ and $D = \{5\}$. Candidate coverage set $S = \{1, 2, 3, 4, 6, 7\}$ is not sufficient to cover D, because the paths $\pi_1 = \langle e, 1, 2, 3, 4, 6, 2, \mathbf{3}, \mathbf{4}, \mathbf{6}\rangle$ and $\pi_2 = \langle e, 1, 2, 3, 4, 6, 2, \mathbf{3}, \mathbf{5}, \mathbf{6}\rangle$ contain the same set of S vertices ($\{1, 2, 3, 4, 6\}$) but π_2 contains $5 \in D$ while π_1 does not. Note the key difference highlighted in **bold**: the instrumentation cannot distinguish a $\langle 3, 4, 6\rangle$ loop iteration from a $\langle 3, 5, 6\rangle$ iteration. In terms of an ambiguous triangle, we have

$$\alpha = 3 \qquad Y = \{1, 2, 3, 4, 6\}$$
$$\beta = 6 \qquad \langle 3, 5\rangle \in P_{\alpha d}$$
$$Y_1.\text{vertices} = \{1, 2, 3, 4, 6\} \qquad \langle 3, 4, 6\rangle \in P_{\alpha\beta}$$
$$Y_2.\text{vertices} = \{2, 3, 4, 6\} \qquad \langle 5, 6\rangle \in P_{d\beta}$$

Again, we see the exact same ambiguity. Because of observations on prior and/or future loop iterations (Y_1 and Y_2), the execution of vertex 4 does not preclude the execution of vertex 5, and we have an ambiguous triangle formed from subpaths $\langle 3, 4, 6\rangle$ and $\langle 3, 5, 6\rangle$.

Consider a few special cases. If $S \supseteq D$, then S is always a coverage set of D, as no possible vertex for d exists (i.e., $D \setminus S = \emptyset$). This aligns with Section 2.3's claim that if $D \subseteq I$ (the instrumentable set), then I itself is a coverage set of D. If $S = \emptyset$, then S is a coverage set of D iff $\forall d \in D$, d occurs on either all or no paths from the entry to any termination point. In this case, $\alpha = e$, $\beta \in X$, and $Y \cap S = \emptyset$. Thus, by the definition of $p_{\alpha d}$, $p_{\alpha\beta}$, and $p_{d\beta}$, S is not a coverage set of D if d may or may not occur on paths from e to $\beta \in X$.

All of our approaches assume that I is a coverage set of D. In practice, one might consider cases where this is untrue (and ask for *maximal* coverage given a limited I set). Fortunately, because each $d \in D$ is independent, we can find the maximal $D' \subseteq D$ that I can possibly cover by letting $D' = \{d \in D$ such that I is a coverage set of $\{d\}\}$.

3.2 Optimal MILP Formulation

As shown in Section 2.5, obtaining an optimal solution to the Customized Coverage Probing Problem is NP-hard. Using the characterization in Section 3.1, we are able to construct a 0–1 mixed integer linear optimization problem (MILP) whose solution identifies the optimal coverage set. Due to space limitations and the intractability of obtaining an optimal solution (see Section 4), we give only an overview of the formulation; the complete MILP model can be found in our companion technical report [31].

To begin, we can define all possible ambiguous triangles for all possible sets $S \subseteq I$ as the set of triples of vertices $\mathcal{T} = \{(\alpha, \beta, d) \in (I \cup \{e\}) \times (I \cup X) \times D\}$. Then, for each $(\alpha, \beta, d) \in \mathcal{T}$ we define the additional set of vertices

$$Y_{\alpha\beta d} = \bigcup_{x \in X, \pi \in (e \xrightarrow{\notin \{d\}} \alpha) \cup (\beta \xrightarrow{\notin \{d\}} x)} V(\pi).$$

For each (α, β, d) triple, the set $Y_{\alpha\beta d}$ corresponds exactly to the Y set from Section 3.1. That is, it contains all vertices along d-free paths from e to α (Y_1) or β to some termination point (Y_2). We can compute this set by checking basic graph connectivity. These sets are provided as input to the MILP model.

The goal is to find S, a minimal-cost coverage set of D. We first introduce the binary selection variables

$$z_i = 1 \text{ iff } i \in S$$

to represent the selected coverage set. Next, we use five sets of binary variables, one for each path set in the coverage set characterization from Section 3.1, to force the associated set to be empty:

$$s_{\alpha d} = 1 \text{ will imply that } e \xrightarrow{\notin \{d\}} \alpha = \emptyset$$
$$t_{\beta d} = 1 \text{ will imply that } \beta \xrightarrow{\notin \{d\}} x = \emptyset \ \forall x \in X \setminus \{d\}$$
$$u_{\alpha\beta d} = 1 \text{ will imply that } \alpha \xrightarrow{\notin S \setminus Y_{\alpha\beta d}} d = \emptyset$$
$$v_{\alpha\beta d} = 1 \text{ will imply that } \alpha \xrightarrow{\notin (S \setminus Y_{\alpha\beta d}) \cup \{d\}} \beta = \emptyset$$
$$w_{\alpha\beta d} = 1 \text{ will imply that } d \xrightarrow{\notin S \setminus Y_{\alpha\beta d}} \beta = \emptyset$$

The model is constructed as a network flow problem. Each of the above variables is accompanied by a set of constraints that force the non-existence of the respective path by constraining the dual flow equations induced by the program's control-flow graph and basic linear programming duality theory (Farkas' Lemma) [11].

Recall from Section 3.1 that S is a coverage set of D if and only if at least one of these five sets of paths is empty for all $(\alpha, \beta, d) \in \mathcal{T}$. These paths are only relevant when $z_d = 0$, because instrumented vertices are always observed. To force this condition, we thus introduce the constraint:

$$s_{\alpha d} + t_{\beta d} + u_{\alpha\beta d} + v_{\alpha\beta d} + w_{\alpha\beta d} \geq (1 - z_d) \ \forall (\alpha, \beta, d) \in \mathcal{T}$$

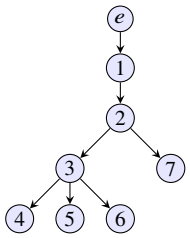


Figure 3: Dominator tree for Fig. 1a

global: *willInst*, the set of vertices that will be probed
global: *willCover*, the set of vertices for which coverage information will be available

input: $G = (V, E)$, a single-function control-flow graph
input: $e \in V$, the entry vertex
input: $I \subseteq V$, vertices that may be probed
input: $c : V \mapsto \mathbb{R}^+$, costs for vertices
input: $D \subseteq V$, vertices with desired coverage
input: $X \subseteq V$, possible ending vertices
output: *willInst* $\subseteq I$, a coverage set of D

T = dominator tree for G, with entry vertex e;
ord_T = any bottom-up ordering of T.vertices;

willInst = $\emptyset$;
willCover = $\emptyset$;
canCover = $\emptyset$;
needInst = $\emptyset$;
foreach v in *ord_T* **do**
  coveredChildren = T.children_of(v) $\cap$ *willCover*;
  if ¬exitWithout(*v, coveredChildren, X*) **then**
    willCover ∪={v};
    canCover ∪={v};
  else
    canCoverChildren = T.children_of(v) $\cap$ *canCover*;
    vNeedInst = exitWithout(*v, canCoverChildren, X*);
    if $v \in I \vee \neg$*vNeedInst* **then** *canCover* ∪={v} ;
    if *vNeedInst* **then** *needInst* ∪={v} ;
    if $v \in D$ **then**
      if $v \notin$ *canCover* **then return** FAIL ;
      else cover(*v, canCover, needInst, T, X, c*) ;

return *willInst*;

Figure 4: Dominator-based approximation

In the end, our objective is to minimize cost

$$\sum_{i \in V} c_i z_i$$

subject to the above forcing constraint on program paths, and the relevant constraints for each of the five classes of paths.

This approach is guaranteed to provide a provably optimal coverage set, but unfortunately is too slow in practice. In fact, we were only able to evaluate the fully optimal approach on our smallest test subjects (see Section 4). There are a number of reasons for this. The formulation requires pre-computation of the sets $Y_{\alpha\beta d}$, which is quartic in the number of vertices in G. Further, even with powerful commercial software, solving a large-scale MILP still relies on enumerating an often large branch-and-bound search tree. Fortunately, safe and fast approximations of the optimal result are possible.

3.3 An Inexpensive Approximation

Several prior approaches optimize coverage probes by using the dominance relation among basic blocks [1, 37]. A basic block v dominates a basic block w if and only if $e \xrightarrow{\notin\{v\}} w = \emptyset$. Immediate dominance relations for any single-entry directed graph form a tree, and algorithms for computing dominators are well-known [3, 20]. Figure 3 shows the dominator tree for the example from Fig. 1a.

Function exitWithout(*v, children, X*)
  input: v, the vertex possibly covered
  input: *children*, a subset of v's immediate-dominator children that may provide coverage information for v
  input: X, possible ending vertices
  return $\exists x$ such that $x \in X$ and v does not dominate x and $v \xrightarrow{\notin children} x \neq \emptyset$;

Figure 5: Test for an exit bypassing dominator children

In this section, we develop an inexpensive approximation algorithm based on dominator information, rather than the sufficiency condition from Section 3.1. Our approach performs a bottom-up traversal of the dominator tree, "covering" a block's subtrees only as necessitated by the desired set, D. This approach is inspired by Agrawal [1] and Tikir and Hollingsworth [37], but supports customized coverage. Most accurately, our approach generalizes these prior approaches, which could be considered special cases of our algorithm for only complete executions [1] and full coverage [1, 37].

A vertex, v, can be "covered" (i.e., guaranteed accurate coverage information for any execution) in two possible ways. First, v itself may be instrumented, so that its coverage is observed directly. Second, we might instrument an appropriate subset of v's dominator-tree descendants such that all executions through v must execute at least one vertex in the descendant set. Clearly, for this approximation, we must instrument all leaves of the tree. Internal block v must be instrumented only if v's dominator-tree children cannot cover v. In our case, v is instrumented if a path exists in G from v to some $x \in X$ that bypasses all of v's covered children in the dominator tree (and such that v does not dominate x). By the definition of dominance, any time a descendant of v executes (including a crashing execution), it implies the execution of v. Intuitively, if the program can halt after executing v without an observation implying v's execution, then v's coverage data is unknown on some execution.

For example, consider vertex 3 in Fig. 3, and a crash $x = 7$ (i.e., the program halts in block 7). Then the subset of dominator children $\{4, 5\}$ is sufficient to cover 3: in Fig. 1, all paths from 3 to 7 must pass through some element of $\{4, 5\}$. Likewise, $\{6\}$ would also cover 3: any CFG path from 3 to 7 must pass through 6. Of course, both alternatives assume that I includes the necessary instrumentation points. If I disallows both $\{4, 5\}$ and $\{6\}$ as instrumentation plans, then 3 can only be covered by direct instrumentation of 3 itself.

Figure 4 details our algorithm. The global set *willInst* builds up the final result: the set of basic blocks to be probed for coverage. The global set *willCover* tracks which blocks will be guaranteed to have accurate coverage information available. Hence, as vertices are added to *willInst*, *willCover* is updated to reflect the newly covered nodes. The overall goal is to make $D \subseteq$ *willCover*. First, we compute T, the dominator tree of G, and any bottom-up ordering of T's vertices, *ord_T*. Then, we iterate over each vertex in *ord_T*, adding those vertices from I that require instrumentation to the set *willInst*. During this iteration, we discover which vertices can only be covered via direct instrumentation (stored in set *needInst*), and which could possibly be covered either via direct instrumentation or via their dominator descendants (stored in set *canCover*).

In the loop, we first find v's dominator children that are already covered (*coveredChildren*). If these vertices are already sufficient to cover v (no path exists from v to an exit or crash bypassing

```
Procedure cover(v, canCover, needInst, T, X, c)
    input: v ∈ canCover, the vertex to cover
    input: canCover, a pre-computed set of vertices that could be
           covered
    input: needInst, a pre-computed set of vertices whose coverage
           information can only be determined by direct probing
    input: T, the dominator tree for the function containing v
    input: X, possible ending vertices
    input: c, costs for vertices
    if v ∈ needInst then
        willInst ∪={v};
    else
        canCoverChildren = T.children_of(v) ∩ canCover;
        assert ¬exitWithout(v, canCoverChildren, X);
        removableChildren = canCoverChildren \ willCover;
        foreach w in removableChildren ordered by c do
            if exitWithout(v, canCoverChildren \ {w}, X) then
                cover(w, canCover, needInst, T, X, c);
            else
                canCoverChildren \={w};
    willCover ∪={v};
```

Figure 6: Cover a dominator tree vertex

the covered children, as defined in function exitWithout() from Fig. 5), then v is added to *willCover* and *canCover*. Otherwise, we gather all of v's dominator children that possibly could be covered (*canCoverChildren*). Because we process T bottom-up, *canCover* already contains all of v's dominator children that could possibly be covered. If *canCoverChildren* is insufficient to cover v, then v is added to *needInst*. If v can be covered by its dominator children or can be directly instrumented, v is added to *canCover*. At this point, if $v \in D$, we want to find a cheap coverage set for v. However, note that this approximation assumes that a vertex can only be covered by its dominator descendants, which is not always true. If we desire coverage for v but $v \notin$ *canCover*, then the algorithm fails to find a coverage set for D. This situation is rare in practice, and another approach (see Section 3.4) could reduce the size of I in this case. If $v \in$ *canCover*, we find a coverage set for v via a call to cover().

Procedure cover() in Fig. 6 walks back down the dominator tree in order to cheaply cover vertex v. First, if v cannot be covered by its dominator children (i.e., $v \in$ *needInst*), then we instrument v to obtain its coverage data. Otherwise, we iterate over all of v's dominator children that can be covered, sorted by cost to try to avoid instrumenting the costliest vertices. For each child w, if w is already covered, we skip it. Otherwise, if w is necessary to cover v (as determined by the call to exitWithout()), we must recursively cover w. After the completion of cover(), v is covered either via direct instrumentation, or via calls to cover() on its descendants. Thus, we pass each vertex as argument v to cover() at most once, since v is added to *willCover* at the conclusion of cover(), and will be excluded from *removableChildren* in future calls.

Our approach is most similar to that of Tikir and Hollingsworth [37], who instrument a basic block, v, whenever v is either a leaf vertex in the dominator tree, or has an outgoing edge (in G) to a block that v does not dominate. This is equivalent to our approach in the un-customized special case of $I = D = X = V$, since any such outgoing edge from v targets a possible halting location. However, our approach handles the full range of input from Section 2, allowing us to optimize coverage with far more degrees of flexibility. We look for paths to any non-dominated termination point (Fig. 5), and only cover vertices where necessitated by D (Fig. 6).

```
input: I ⊆ V, vertices that may be probed
input: c : V ↦ ℝ⁺, costs for vertices
input: D ⊆ V, vertices with desired coverage
output: S ⊆ I, a locally optimal coverage set of D
assert I is a coverage set of D;
S = copy(I);
tryRemove = sort I by c;
foreach i in tryRemove do
    if S \ {i} is a coverage set of D then
        S \={i};
return S;
```

Figure 7: Locally optimal approximation

3.4 Locally Optimal Approximation

The approach in Section 3.3 is computationally inexpensive: it calls cover() on each block at most once, and, therefore traverses each dominator tree vertex at most twice. However, it provides no guarantees on the optimality of the obtained *willInst* set. In fact, as noted in Section 3.3, it is possible that the dominator-based approximation will be unable to find any coverage set $S \subseteq I$, even if at least one such set exists. This is the "**return** FAIL" case in Fig. 4.

We can compute a locally optimal coverage set in polynomial time by iteratively testing smaller-and-smaller candidate coverage sets via the conditions in Section 3.1. By these conditions, a candidate coverage set S can clearly be checked in polynomial time. For each (α, β, d) triple arising from our current choice of S, we:

1. compute Y, which requires two depth-first or breadth-first search passes (one to gather all possible vertices along paths $e \xrightarrow{\notin\{d\}} \alpha$, and one to gather vertices along paths $\beta \xrightarrow{\notin\{d\}} x$);
2. check for the existence of any path $\alpha \xrightarrow{\notin S \setminus Y} d$;
3. check for the existence of any path $\alpha \xrightarrow{\notin (S \setminus Y) \cup \{d\}} \beta$; and
4. check for the existence of any path $d \xrightarrow{\notin S \setminus Y} \beta$.

Each of the three connected-excluding tests again requires a single depth-first or breadth-first search. If, for any (α, β, d) triple, each of the collected vertex sets from Item 1 are non-empty and a path exists for all Items 2 to 4, then S is *not* a coverage set of D.

A coverage set S is locally minimal with respect to D when S is a coverage set of D, and $\forall S' \subset S$, S' is not a coverage set of D. Figure 7 gives the direct approach. We begin with $S = I$ (a coverage set of D by assumption), and iteratively attempt to remove each element of S. Removing vertices from S can never cause S to cover *more* vertices. Thus, if S is a coverage set of D, then $\forall S^{\uparrow} \supset S$, $S^{\uparrow}$ is also a coverage set of D. Contrapositively, if S is *not* a coverage set of D, then $\forall S^{\downarrow} \subset S$, $S^{\downarrow}$ is not a coverage set of D either.

This approach has polynomial time complexity, but performs redundant computation, and is too inefficient for practical use. We improve performance using a number of optimizations and heuristics. We begin by reducing our initial I set by a call to our dominator-based approximation (Fig. 4). If this approximation returns FAIL, then I cannot be proven a coverage set of D using only dominance relations, and we begin with the full user-specified I set. As Fig. 7 shows, we heuristically attempt to first remove the costliest vertices from S. We pre-compute $V(Y_1)$ for each (α, d) pair, and $V(Y_2)$ for each (β, d) pair, as these sets are not dependent on the choice of S. We also perform substantial pruning of possible (α, β, d) triples. For example, as Fig. 2 illustrates, all possible α vertices must precede d,

Table 1: Evaluated applications, ordered by size. Compilation times are scaled relative to 1 for standard Clang with no instrumentation. "–" marks compilations that did not complete within 3 hours.

				Relative Compilation Time					
				Basic Block Coverage			Call Coverage		
Application	Description	Versions	Mean LOC	None	Dominators	Local	None	Dominators	Local
tcas	Siemens	1	173	1.3	1.3	1.3	1.3	1.3	1.3
schedule2	Siemens	1	373	1.4	1.4	2.0	1.3	1.3	1.3
schedule	Siemens	1	413	1.4	1.4	5.1	1.3	1.3	1.5
replace	Siemens	1	563	1.4	1.4	43.2	1.3	1.3	1.3
tot_info	Siemens	1	564	1.3	1.3	14.2	1.3	1.3	1.4
print_tokens2	Siemens	1	568	1.3	1.3	5.4	1.3	1.3	1.3
print_tokens	Siemens	1	727	1.4	1.4	360.6	1.3	1.3	1.7
ccrypt	Linux utility	1	5,280	1.5	1.5	5.1	1.4	1.4	1.6
gzip	Linux utility	5	8,114	1.7	1.6	3,157.8	1.4	1.4	64.7
space	ADL interpreter	1	9,563	1.6	1.6	24.8	1.5	1.5	1.6
exif	Linux utility	1	10,611	1.5	1.5	22.2	1.5	1.4	12.3
bc	Linux utility	1	14,292	1.6	1.6	365.5	1.5	1.5	18.7
sed	Linux utility	7	14,314	2.0	1.8	–	1.5	1.5	3,744.4
flex	Linux utility	5	14,946	2.1	1.8	–	1.6	1.6	–
grep	Linux utility	5	15,460	1.9	1.7	–	1.4	1.4	12.1
bash	Linux shell	6	80,443	1.5	1.5	–	1.5	1.5	–
gcc	C compiler	1	222,196	1.8	1.7	–	1.5	1.5	–

and all β must follow d. Finally, we find that, in practice, ambiguous triangles tend to exist in close proximity to the un-covered $d \in D$. In other words, the length of paths in $\alpha \xrightarrow{\notin S \setminus Y} d$, $\alpha \xrightarrow{\notin (S \setminus Y) \cup \{d\}} \beta$, and $d \xrightarrow{\notin S \setminus Y} \beta$ tends to be short. Thus, we prioritize testing α and β vertices crossing the fewest edges from d.

Overall, though, our approach does not fundamentally differ from Fig. 7. We are actively searching for optimizations and heuristics that increase performance, particularly for large, complex CFGs. Recall that the approach is approximate; any solution S contains no unnecessary blocks to cover D, but other less-costly instrumentation plans may exist. Thus, as stated, in the context of our MILP from Section 3.2, this approach results in a locally optimal solution.

3.5 Recovering Coverage Data

Extracting desired coverage data from gathered probes can be somewhat complex in cases of aggressive probe optimization. For most prior work in coverage optimization [1, 37], full coverage data can be derived using nothing more than gathered coverage data and a function's dominator/post-dominator trees. However, our approaches (except for that in Section 3.3) use more complex reasoning. Fortunately, our prior research [30] recovers complete coverage information based on incomplete data. The data we collect here can be fed directly into our existing recovery algorithms with no changes to the latter whatsoever.

4. EVALUATION

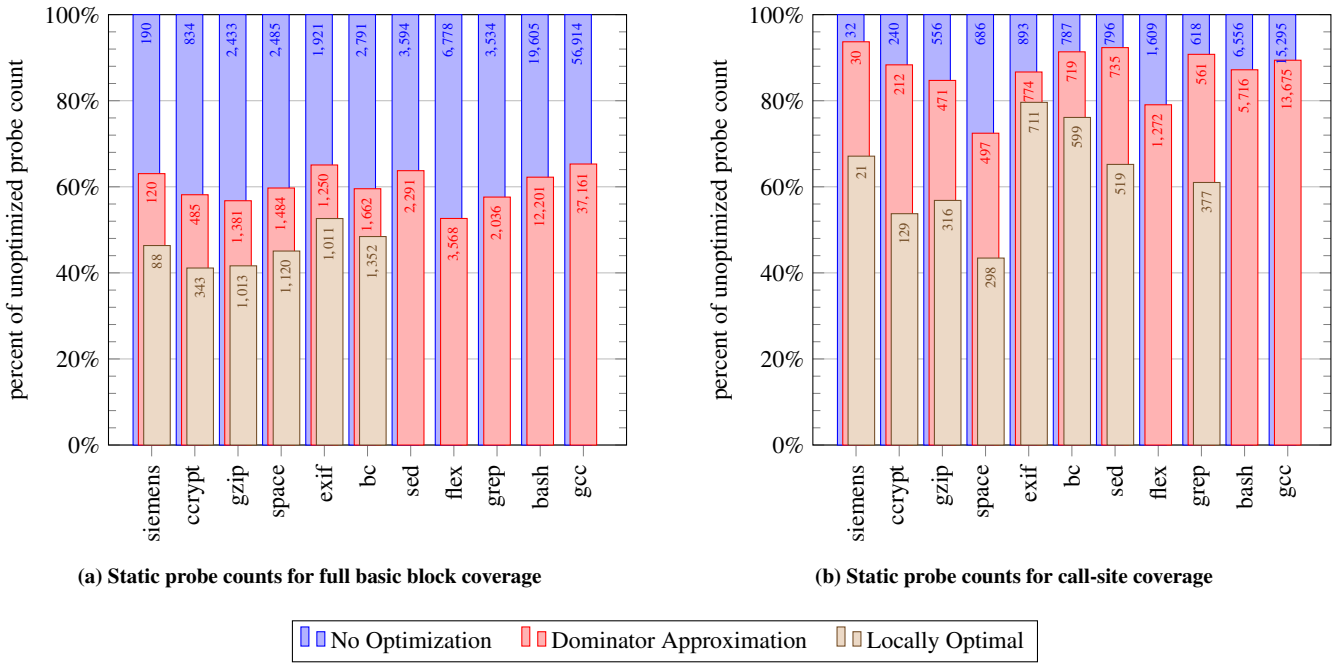
We evaluated our techniques across a wide variety of C benchmarks. Our evaluations assess the efficiency of our instrumentation (as compile-time overhead to optimize coverage probes) and generated probing schemes (as probe counts and run-time overhead).

4.1 Experimental Design

We implemented the techniques described in Section 3 for C/C++ programs. We extended our existing instrumenting compiler csi-cc [29] built with Clang/LLVM 3.5 [19]. csi-cc already has support for maintaining in-memory coverage data post-crash. We use LLVM's built-in `BlockFrequency` analyses to determine costs (c_i for each $i \in I$) as input to our approach. This statically approximates the execution frequency of each block, but is realistic since even a run-time profile approximates post-deployment behavior. We ran two sets of experiments. First, we optimized for full statement coverage: $I = D = V$. Second, we optimized to gather coverage at call sites: $I = D =$ {basic blocks containing at least one call site}. Call-site coverage is an example of customized coverage that cannot be optimized by any prior approach. Note the slight difference from call-site coverage as described in Section 2.4, where we propose allowing instrumentation anywhere (i.e., $I = V$). In practice, we have found that LLVM's static cost model is not always a good representation of run-time costs, which means that our approaches may be driven to choose more expensive instrumentation plans. By setting $I = D$, we ensure that our optimizations can only remove probes; they cannot, for example, cover basic block b by inserting new probes into b's dominator descendants (present in I but absent from D) to assure coverage of b. Note that inaccurate cost data is a threat to run-time efficiency, but never to correctness. That is, our approaches can never select a result S that is not a coverage set of D, even if S results in suboptimal run-time performance. In all cases, we optimize coverage assuming programs may crash at any statement: $X = V$. We ran all experiments on a quad-core Intel Core i5-3450 CPU clocked at 3.10 GHz with 32 GB of RAM and running Red Hat Enterprise Linux 6.7.

Table 1 provides details for our subject programs. We obtained most of these applications from the Software-artifact Infrastructure Repository [12, 35]. The exceptions are bc, ccrypt, exif, and gcc: these are real-world programs. We ran experiments over the non-faulty builds of most applications; gcc and exif used builds with a known fault. Some of the applications had multiple versions as indicated by the "Versions" column of Table 1.

Note that we exclude fully optimal coverage results from all experiments. Our implementation of the MILP formulation from Section 3.2 either exceeds our compilation time limit (3 hours) or runs out of memory for all but a selection of the small Siemens



(a) Static probe counts for full basic block coverage

(b) Static probe counts for call-site coverage

Figure 8: Coverage probe counts. All bars are scaled to 100% for no optimization to show how much relative reduction the locally optimal and dominator-based approximations achieve. Numbers within each bar are mean counts without scaling. For example, at the left edge of Fig. 8a, the Siemens benchmarks average 190 probes with no optimization, but just 120 probes with the dominator-based approximation: 70 probes have been optimized away. In relative terms, these Siemens benchmarks have just 63% as many probes with dominator-based optimization as they do with no optimization.

benchmarks. Our locally optimal implementation from Section 3.4 compiles 12 of our 17 benchmarks for full statement coverage, and 14 benchmarks for call-site coverage, within the time limit.

4.2 Optimization and Compile Time

For each version of each application, we first measured the wall-clock time to perform each of our optimization approaches and instrument the program. These results are shown in Table 1, relative to a base build with "`clang -O3`": a value of 1.0 indicates no compilation-time overhead. We built each application version at least three times, and divided by base compilation time. We then took the geometric mean to aggregate across all versions (to avoid over-representing specific versions). The "None" columns indicate compilation overhead for instrumenting all $i \in I$ for the selected option. The "Dominators" columns show overhead for the dominator-based approximation from Section 3.3. The "Local" columns show overhead to obtain a locally minimal solution. In all cases, we instrumented functions to gather *local* coverage data (see Section 2.4), storing coverage data within the program stack at run time. Storing coverage data in-memory results in substantially smaller absolute overheads, and is common in practice [4, 29].

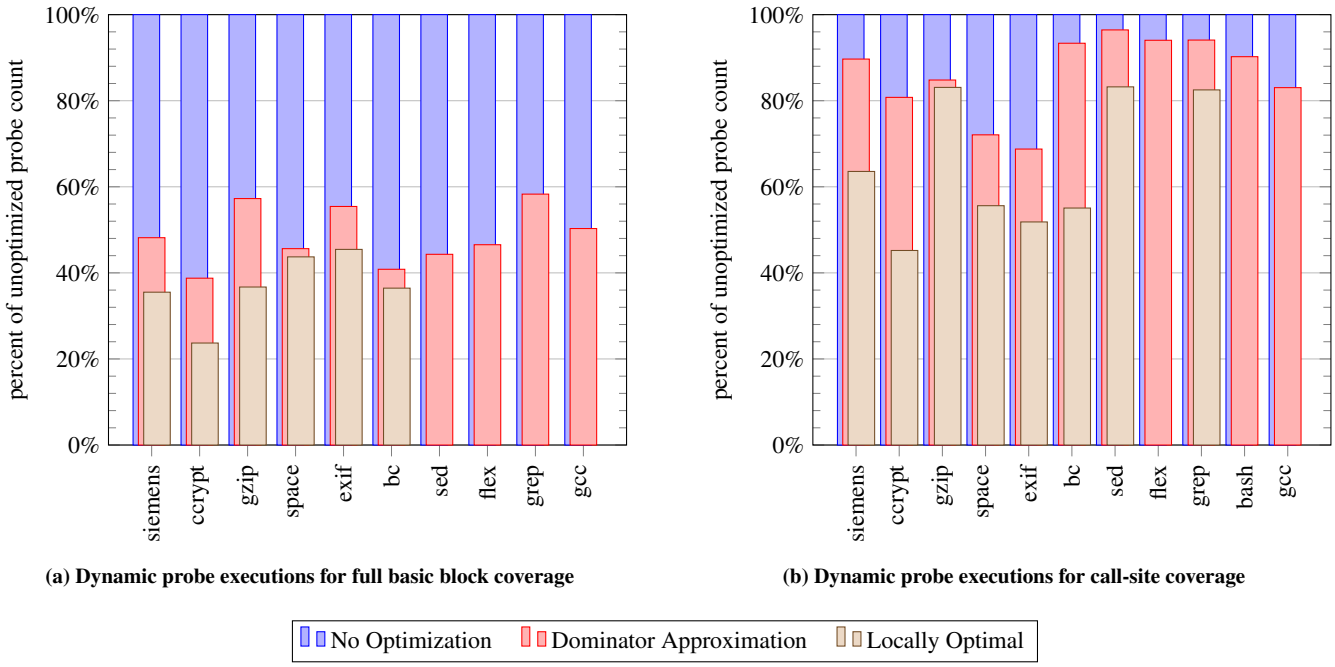
The results are grouped into statement coverage results (gathered as basic block coverage) and call-site coverage results. Without optimization, there is a cost of 1.3×–2.1× to compile benchmarks for coverage. However, the dominator-based optimization is extremely inexpensive, often saving time over full instrumentation. We believe that the extra cost results from generating and inserting the required probing code. The overhead of locally minimal optimization varies greatly between benchmarks. We find that large, complex functions take a disproportionately long time to optimize. For example, gzip's base compile time averages under 2 seconds, but computing a locally optimal solution requires over 1.5 hours, dominated by 3 functions that consume over 90% of the total time. As we discuss in Section 3.4, the approach has polynomial time complexity; however, the description in Section 3.1 indicates that, in the worst case, we must consider all possible (α, β, d) triples in order to prove that a particular $S \subseteq I$ is a coverage set of our desired set D.

Restricting the set of desired blocks, D, reduces the number of (α, β, d) triples considered by our locally optimal approach, and should reduce optimization time. Our call-site coverage compilation results confirm this; we find substantially smaller compile-time overheads when compiling for coverage only at call sites. For example, while gzip's compile time increases from 1.5 seconds to just over 2 minutes, this is far below the 3,000× increase we see for optimizing full statement coverage. These improvements allow us to compute locally optimal solutions for two additional benchmarks (grep and sed). However, 3 benchmarks still do not complete compilation, and sed exhibits a 3,744× slowdown; thus, scalability remains a concern for our locally optimal formulation.

4.3 Static Probe Counts

We also gathered the total number of probes inserted by each approach, to examine the static reduction in probe insertions for our optimizations. We again took the mean of probe counts for different versions of each application. We also aggregated results for all Siemens applications to simplify presentation. Probe reductions are very similar across all Siemens benchmarks of non-trivial size.

Figure 8a shows results for full statement coverage. As noted in Section 3.3, since $I = D = X = V$ for full statement coverage, our dominator-based approximation is equivalent to the optimizations of Tikir and Hollingsworth [37] in this specific scenario. Stacked bars indicate the percentage of the unoptimized probes still included after the specified optimization. Hence, for example, bc with the dominator-based approximation reduces the probe count by approx-



(a) Dynamic probe executions for full basic block coverage

(b) Dynamic probe executions for call-site coverage

Figure 9: Dynamic probe executions. All bars are scaled to 100% for no optimization to show how much relative reduction the locally optimal and dominator-based approximations achieve. Mean counts without scaling are omitted due to space limitations.

imately 40% relative to unoptimized instrumentation. The locally optimal approach further reduces the probe count, cutting the remaining probes to just below half of the original set. Those applications that cannot be compiled within our time limit exclude locally optimal results in the figure. Overall, probe reductions are substantial. Our dominator-based approximation is inexpensive, but still reduces probe counts by over 40% on average. The locally optimal approach is substantially more expensive (as seen in Table 1), but further reduces necessary instrumentation to just 44% of unoptimized instrumentation for completed benchmarks, on average.

The main thrust of our approaches, however, comes in their ability to optimize instrumentation based on *customized* coverage requirements. Figure 8b presents results for call-site coverage. Note that prior work cannot optimize coverage probes in this scenario. Here, the unoptimized instrumentation set is much more selective. Reductions are smaller across all benchmarks, but, for most applications, we still see substantially less instrumentation. The dominator-based approximation reduces probe counts by up to 28% (space), and averages a 15% reduction across all benchmarks. The benefits of the locally optimal approach are very pronounced. For those applications that completed local optimization, we see an average further reduction of 30% from the dominator-based approximation, with total reductions as high as 57% (space, relative to unoptimized).

4.4 Dynamic Probe Counts

With the completed builds, we then gathered the total number of probe executions at run time to assess the *dynamic* impact of optimizations. We ran each application through its corresponding test suite. We gathered the count for each trial, and computed the percentage reduction for each level of optimization. We took the arithmetic mean to aggregate across each complete test suite, and aggregated the resulting values across all versions of each application (to avoid over-representing specific versions or long-running test cases). We again aggregated results for the Siemens applications, which exhibit similar run-time performance.

Figure 9a shows dynamic probe execution reductions for full statement coverage. We excluded one test case for gzip that exceeds our 1-hour timeout for extracting probe counts. We omit bash results, as bash's test suite is highly sensitive to our probe counting infrastructure with dense statement coverage. Stacked bars are scaled to the number of executed probes for the unoptimized variant. Again, $I = D = X = V$, and our dominator-based approximation is equivalent to Tikir and Hollingsworth [37]. For all of the applications, even this approximation results in a substantial drop in overheads; in fact, all applications lose at least 60% of their probe executions, while simultaneously shrinking compile time per Table 1. ccrypt sees the largest reduction, executing just 39% of the unoptimized probe count. Although expensive to compute, overhead reductions from further reducing probes via the locally optimal formulation are sometimes substantial. For example, after the dominator-based approximation reduces gzip's probe executions by 43%, our locally optimal approach removes more probes, reducing probe executions to just 37% relative to uninstrumented code; ccrypt is reduced to just 24% of the unoptimized count. Of course, as mentioned earlier, this run-time performance may come at a cost: gzip's compile time increases from seconds to hours when moving to a locally optimal solution. Overall, the performance of the dominator-based approximation is quite impressive. Our locally optimal approach presents a significant trade-off: it does often remove significantly more probes (see Fig. 8a), but at a very high compilation cost (see Table 1).

Figure 9b presents call-site coverage results. We exclude 10 (out of 1061) bash test cases due to time-out. Here, unoptimized probe executions are much smaller, and reductions from the dominator-based approximation are less pronounced. Nevertheless, some applications see significant benefit. For example, exif and space both execute just 70% of the unoptimized probe counts. Our locally optimal approach, however, is very impressive. While some applications see less benefit (e.g., gzip), many applications see enormous reductions. For example, bc and ccrypt both reduce probe executions by 40% beyond the reductions of the dominator-based approximation.

We also assessed the statistical significance of the results from Fig. 9. We conducted a Wilcoxon signed-rank test between test cases with each level of optimization. For all results, we find sufficient evidence ($p < 0.01$) to reject the null hypothesis that our optimizations have no effect on dynamic probe executions.

4.5 Running Time

The results from Figure 9 do not depend on probe costs, but real impacts on running time depend on the cost to execute each probe. We measured execution times of each program's test suite, measuring overheads relative to "`clang -O3`" as a baseline. We used inline, in-memory probes, which impose far smaller overheads than would be seen if data were occasionally flushed to disk or if probes required function calls to record data. Even so, we observe significant reductions, particularly with statement coverage. Mean overhead for the non-trivial benchmarks shrinks from 7.2% to 3.6% using the dominator-based approximation. Although expensive to compute, the locally optimal formulation can provide substantial additional benefit. For example, the dominator-based approximation reduces gzip's overhead from 11.6% to 5.7%; our locally optimal formulation reduces this further to just 3.4% relative to uninstrumented code. For call-site coverage, our optimizations did not measurably reduce running time. This is surprising, given the reductions shown in Fig. 9b. However, our in-memory binary probes are already quite fast, and the majority of our test cases do not run long enough to exhibit large overheads. If probes were more costly, we would expect much more pronounced results. Further, the larger numbers of probes used for statement coverage show a correspondingly larger optimization benefit. We again used a Wilcoxon test, and found our statement coverage reductions to be statistically significant ($p < 0.01$ for each non-zero overhead reduction).

Overall, our results show we can significantly reduce the static and dynamic cost of customized coverage instrumentation. We specifically evaluate customized coverage of call sites, where required coverage information is already substantially reduced. Even so, our techniques reduce static probe counts by as much as 57% (space in Fig. 8b), and dynamic probe executions by up to 55% (ccrypt in Fig. 9b). Even when the absolute numbers are not large, these reductions should not be discounted: these may be very important in systems with real-time requirements or for deployed software.

5. RELATED WORK

Closely related work optimizes placement of binarized coverage probes. Agrawal [1] optimizes probes by forming "superblocks" from sets of basic blocks based on dominance and post-dominance relations. Later, Agrawal [2] extends this work to interprocedural dominance relations; Li et al. [21] and Xu et al. [39] extend these optimizations beyond superblocks. Tikir and Hollingsworth [37] also optimize coverage probe placement via dominators, but use a faster, simpler approach (and no post-dominance information) for online instrumentation. Our approximation in Section 3.3 is inspired by many of these prior approaches. However, because we support multiple crash points (via input parameter X), we cannot directly take advantage of post-dominator information as in Agrawal. Further, because we allow customization of desired and instrumentable locations, we develop a generalization of these existing approaches, facilitating many coverage optimization scenarios that are not supported by any prior work. Particularly for local coverage data (i.e., without the transformation from Fig. 1b), our approaches can optimize instrumentation more aggressively than any prior approach.

Binarized coverage only needs to observe each probed location one time: once a coverage value becomes true, it stays true. Building on this insight, prior work optimizes coverage gathering via dynamic insertion and deletion of probes [10, 16, 26, 27, 33, 37]. These techniques are complimentary to our own: we optimize *where* to insert probes, while such techniques address *how* and *when* to insert probes. Many commercial tools gather program coverage over test suites (e.g., [4, 9, 14, 15]). These tools gather complete program coverage, whereas our work allows a developer to *focus* tracing to reduce overheads and/or limit possible instrumentation.

Prior work has optimized instrumentation to gather *frequency counts* of statements or edges, often to identify program "hot spots." Knuth and Stevenson [18] optimize frequency counter placement for program statements, and Knuth [17] optimizes instrumentation for edge counts. Ball and Larus [5] formalize these classic approaches, and generalize the counting problems for vertices and edges. While these classic approaches run in polynomial time, their solutions cannot be used for binarized instrumentation: they rely on Kirchoff's current law, which does not hold of binarized indicators. Further, many use cases that we consider would not make use of the more detailed count information. Coverage data can obviously be derived from count data, but the cost of gathering counts is higher than the cost of gathering coverage. In contexts where counts are not required, binarized coverage has a number of advantages: (1) each probe is less expensive to execute, (2) each probe requires only one bit of storage rather than a larger, overflow-vulnerable integer, (3) instrumentation is easily made thread-safe on all architectures, and (4) probes can be removed after they are first triggered.

Pavlopoulou and Young [34] monitor residual coverage of code missed during testing. The GAMMA project [8, 32] adapts post-deployment instrumentation for data collection aggregated across large user communities; each individual entity only traces a subset of desired data. Our optimizations are directly applicable here, further reducing required coverage probes for each deployed instance.

Prior work optimizes the set of test cases required to achieve a specific coverage criterion [1, 7, 24, 40]. Early approaches to test suite minimization have much in common with approaches to minimizing coverage probes: for example, all can make use of dominance information for implied coverage. However, our problem is different in that desired coverage information must be guaranteed for *any* run, rather than attained from a minimal *set* of runs.

Other work has developed the idea of relative coverage in the context of web services [6, 13, 25]. Our work can also facilitate customizing coverage metrics to context-dependent targets, but deals with optimizing coverage probe placement rather than how to gather and present this data to users of a web service.

6. CONCLUSION

Binarized program coverage information is used in a wide variety of scenarios, from the testing lab to post-deployment monitoring. Different situations yield very different requirements for coverage, as well as different run-time overhead restrictions. We present a system that allows users to specify customized coverage criteria: desired coverage locations, as well as the set of locations that are valid for instrumentation. While we show that the Customized Coverage Probing Problem is NP-hard, we also develop inexpensive approximations, and show that even coarse approximations can lead to significant reductions in instrumentation cost.

7. ACKNOWLEDGMENTS

This research was supported in part by DARPA MUSE award FA8750-14-2-0270 and NSF grants CCF-0953478, CCF-1217582, CCF-1318489, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

8. REFERENCES

[1] H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 25–34, New York, NY, USA, 1994. ACM. URL http://doi.acm.org/10.1145/174675.175935.

[2] H. Agrawal. Efficient coverage testing using global dominator graphs. In W. G. Griswold and S. Horwitz, editors, *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '99, Toulouse, France, September 6, 1999*, pages 11–20. ACM, 1999. URL http://doi.acm.org/10.1145/316158.316166.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Atlassian. Clover, Jan. 2016. URL https://www.atlassian.com/software/clover.

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

[6] C. Bartolini, A. Bertolino, S. G. Elbaum, and E. Marchetti. Whitening SOA testing. In H. van Vliet and V. Issarny, editors, *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 161–170. ACM, 2009. URL http://doi.acm.org/10.1145/1595696.1595721.

[7] A. Bertolino. Unconstrained edges and their application to branch analysis and testing of programs. *Journal of Systems and Software*, 20(2):125–133, 1993. URL http://dx.doi.org/10.1016/0164-1212(93)90004-H.

[8] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '02, pages 2–9, New York, NY, USA, 2002. ACM. URL http://doi.acm.org/10.1145/586094.586099.

[9] Bullseye Testing Technology. BullseyeCoverage, Jan. 2016. URL http://www.bullseye.com/productInfo.html.

[10] K. Chilakamarri and S. G. Elbaum. Leveraging disposable instrumentation to reduce coverage collection overhead. *Softw. Test., Verif. Reliab.*, 16(4):267–288, 2006. URL http://dx.doi.org/10.1002/stvr.347.

[11] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[12] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005. URL http://dx.doi.org/10.1007/s10664-005-3861-2.

[13] M. M. Eler, A. Bertolino, and P. C. Masiero. More testable service compositions by test metadata. In J. Z. Gao, X. Lu, M. Younas, and H. Zhu, editors, *IEEE 6th International Symposium on Service Oriented System Engineering, SOSE 2011, Irvine, CA, USA, December 12-14, 2011*, pages 204–213. IEEE Computer Society, 2011. URL http://dx.doi.org/10.1109/SOSE.2011.6139109.

[14] Free Software Foundation. Gcov: a test coverage program, Jan. 2016. URL https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[15] IBM Rational. PureCoverage, 2003. URL ftp://ftp.software.ibm.com/software/rational/docs/v2003/purecov/index.htm.

[16] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi. Efficient tracing of cold code via bias-free sampling. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 243–254. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/kasikci.

[17] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

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

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

[20] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. URL http://doi.acm.org/10.1145/357062.357071.

[21] J. J. Li, D. M. Weiss, and H. Yee. An automatically-generated run-time instrumenter to reduce coverage testing overhead. In *Proceedings of the 3rd International Workshop on Automation of Software Test, AST 2008, Leipzig, Germany, May 11-11, 2008.*, pages 49–56. ACM, 2008. URL http://dx.doi.org/10.1145/1370042.1370054.

[22] B. Liblit. The Cooperative Bug Isolation Project, Jan. 2014. URL http://research.cs.wisc.edu/cbi/.

[23] S. N. Maheshwari. Traversal marker placement problems are NP-complete. Technical Report CU-CS-092-76, University of Colorado, Boulder, July 1976.

[24] M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Software Eng.*, 29(11):974–984, 2003. URL http://dx.doi.org/10.1109/TSE.2003.1245299.

[25] B. Miranda and A. Bertolino. Social coverage for customized test adequacy and selection criteria. In H. Zhu, J. Gao, S. Sinha, and L. Zhang, editors, *9th International Workshop on Automation of Software Test, AST 2014, Hyderabad, India, May 31 - June 1, 2014*, pages 22–28. ACM, 2014. URL http://doi.acm.org/10.1145/2593501.2593505.

[26] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In G. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 156–165. ACM, 2005. URL http://doi.acm.org/10.1145/1062455.1062496.

[27] J. Misurda, B. R. Childers, and M. L. Soffa. Jazz2: a flexible and extensible framework for structural testing in a Java VM. In C. W. Probst and C. Wimmer, editors, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*, pages 81–90. ACM, 2011. URL http://doi.acm.org/10.1145/2093157.2093169.

[28] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 422–431, New York, NY, USA, 1999. ACM. URL http://doi.acm.org/10.1145/302405.302674.

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

[30] P. Ohmann, D. B. Brown, B. Liblit, and T. W. Reps. Recovering execution data from incomplete observations. In H. Xu and W. Binder, editors, *Proceedings of the 13th International Workshop on Dynamic Analysis, WODA 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 19–24. ACM, 2015. URL http://doi.acm.org/10.1145/2823363.2823368.

[31] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Encoding optimal customized coverage instrumentation. Technical Report 1836, Department of Computer Sciences, University of Wisconsin–Madison, Aug. 2016.

[32] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 65–69, New York, NY, USA, 2002. ACM. URL http://doi.acm.org/10.1145/566172.566182.

[33] T. Pankumhang and M. Rutherford. Iterative instrumentation for code coverage in time-sensitive systems. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE, 2015. URL http://dx.doi.org/10.1109/ICST.2015.7102594.

[34] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 277–284. ACM, 1999. URL http://portal.acm.org/citation.cfm?id=302405.302637.

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

[36] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 56–66. IEEE, 2009. URL http://dx.doi.org/10.1109/ICSE.2009.5070508.

[37] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA*, pages 86–96, 2002. URL http://doi.acm.org/10.1145/566172.566186.

[38] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In H. Mulder and M. K. Farrens, editors, *Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, California, USA, November 30 - December 2, 1994*, pages 1–11. ACM/IEEE, 1994. URL http://dx.doi.org/10.1109/MICRO.1994.717399.

[39] X. Xu, Y. Chen, W. E. Wong, and D. Guo. Vnm: A novel method to reduce the overhead of program instrumentation. In *Proceedings of the 2009 WRI World Congress on Software Engineering - Volume 04*, WCSE '09, pages 256–260, Washington, DC, USA, 2009. IEEE Computer Society. URL http://dx.doi.org/10.1109/WCSE.2009.315.

[40] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab.*, 22(2):67–120, 2012. URL http://dx.doi.org/10.1002/stv.430.