Probe Distribution Techniques to Profile Events in Deployed Software

Madeline Diep, Myra Cohen, Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{mhardojo, myra, elbaum}@cse.unl.edu

Abstract

Profiling deployed software provides valuable insights for quality improvement activities. The probes required for profiling, however, can cause an unacceptable performance overhead for users. In previous work we have shown that significant overhead reduction can be achieved, with limited information loss, through the distribution of probes across deployed instances. However, existing techniques for probe distribution are designed to profile simple events. In this paper we present a set of techniques for probe distribution to enable the profiling of complex events that require multiple probes and that share probes with other events. Our evaluation of the new techniques indicates that, for tight overhead bounds, techniques that produce a balanced event allocation can retain significantly more field information.

1. Introduction

Software profiling consists of observing, gathering, and analyzing data to characterize a program's behavior. Profiling deployed software is valuable for providing insights into how software is utilized by its users [17], which configurations are being employed in the field [15], what engineering assumptions formed during development may not hold in the field or may hold only under more constrained circumstances [5], where the current in-house validation activities are lacking [6], and which scenarios most likely led to a failure state (e.g., Traceback [11], MSN Error API [16]).

Profiling usually requires instrumentation of the program, that is, the addition of probes to enable the examination of the program's run-time behavior. As such, the act of profiling penalizes the targeted software with execution overhead. For example, Liblit *et al.* report an overhead ranging from 2% to 181% on various benchmarks profiling assertion invocations [13], and in the study we will be presenting, fully instrumenting a target program to capture call-chains causes a 150% overhead in execution time.

The magnitude of the overhead depends, at least to some extent, on the number, the location, and the type of inserted profiling probes. To lower the profiling overhead, instrumentation techniques may insert only a subset of probes, but do so strategically to reduce data loss. For example, a common instrumentation approach to lower the number of probes consists of skipping probes that capture redundant or inferable data.

Lowering overhead is particularly critical for profiling software in the field where it can have a direct impact on the user. As a result, researchers are attempting to leverage what can potentially be thousands of deployed sites not just to characterize a population of events that are much richer than the ones available in-house, but also to lower the number of inserted probes by distributing them across a pool of sites. Such an approach consists of two main steps: 1) generating multiple program variants, where each variant contains a subset of probes, and where the subset size can be bounded to meet the overhead requirements, and 2) distributing those variants across the user sites [6, 18].

In previous work we have introduced and evaluated several instances of this approach [5]. For example, we have shown that probe distribution techniques relying on simple sampling principles such as stratified sampling can effectively reduce profiling overhead at each user site by up to one order of magnitude. This overhead reduction greatly increases the viability of profiling released software, while still retaining a majority of the field information. We later improved those techniques by utilizing a greedy-sampling process to generate variants with balanced distributions [3].

Although successful, our initial probe distribution techniques assume that the population of events being characterized is made up of relatively simple and independent events (e.g., the execution of a block of code). In practice, however, we would like to profile events such as execution paths, exceptional control flows, or call chains. Profiling one such event may require multiple probes, and some events may be somewhat dependent in that they can share overlapping probes. Furthermore, there may be too many of these events for an engineer to enumerate cost-effectively before deployment. When these issues arise, existing probe distribution techniques can produce unsatisfactory results.

2. A Motivating Example

Call-chains are sequences of method calls. Engineers use call-chains for many purposes: to determine whether a given method can indirectly reach another method, to enumerate the possible paths of execution between a pair of methods, to assist debugging by providing the sequence of method calls executed before a failure, to assist re-engineering activities, to serve as drivers for program testing, or as testing oracles in the presence of program changes [12, 23, 24].

Call-chains can be computed through the static analysis of a program's source code or through the dynamic analysis of the program's execution. The precision of statically computed call-chains relies on the precision of the call graph utilized as input, and although many techniques have been proposed to increase the precision of call graph construction, statically computing call-chains can still be expensive and it is commonly an over-approximation (some of the call chains reported may not be feasible). This limitation becomes more obvious when we try to capture call-site sensitive call chains, that is, call-chains that are differentiated by the particular site within the caller where the invocation to the callee is made.

Call-chains gathered by observing a program's execution are precise (no false positives), but the in-house activities are unlikely to expose a significant number of them. Collecting call-chains executed by a very large number of users in the field can overcome this limitation, providing a richer set of executions that result in a precise yet comprehensive enumeration of the program call-chains. The challenge is to then profile such complex events with minimal overhead while maximizing the information captured in the field.

In this paper we re-define the probe distribution problem across variants, taking into consideration the challenges introduced by complex events such as call-chains, and we present a set of more general techniques for probe distribution. To investigate the efficiency and effectiveness of the new techniques, we compare them against existing probe distribution techniques designed for single probe events through an empirical study. Our findings indicate that for scenarios where the acceptable profiling overhead is small the proposed distribution techniques can make a significant practical difference.

3. Background

Researchers have investigated many ways to enhance the efficiency of profiling techniques such as: (1) performing up-front analysis to optimize the amount of instrumentation [2, 25], (2) sacrificing accuracy by monitoring entities of higher granularity or by sampling program behavior [7, 8], (3) encoding the information to reduce memory and storage requirements [21], and (4) re-targeting the entities that need to be profiled [1, 4]. These mechanisms are created to work in the controlled and small-scale in-house environment.

When profiling deployed software, however, we can also leverage the opportunities introduced by a potentially large pool of users. The following research efforts, for example, focus in that direction: 1) Perpetual Testing, which uses the residual testing technique to reduce instrumentation [19], 2) EDEM, which provides a semi-automated way to collect user-interface feedback from remote sites when the user actions do not meet an expected criterion [10], 3) Skoll, which presents an architecture and a set of tools to distribute different job configurations across users [15], 4) Gamma, which introduces an architecture to distribute and manipulate instrumentation across deployed instances [18], and 5) Bug Isolation, which provides an infrastructure to efficiently collect field data and identify likely failure conditions [13, 14]. Our work is closely related to the last two projects.

It is similar to Gamma in that the overall mechanism to reduce overhead utilizes probe distribution across deployed instances. However, our focus has been on developing techniques to achieve distributions that increase the likelihood of retaining field data [6]. Our work is also related to the bernoulli-sampling used by the Bug-Isolation project since both attempt to reduce overhead through some form of sampling. However, while the bernoulli-sampling approach includes additional instrumentation to determine *when* to sample at run-time, our objective is to determine before deployment *what* to sample in each variant. Our approach aims to produce a balanced "sample" across variants, without incorporating the additional instrumentation required to adjust the sampling at run-time.

The work in this paper builds on our previous studies by considering instrumentation constraints, the association between events and probes, and the relationship between events in terms of probe sharing. Techniques incorporating this factors are studied on a truly deployed non-trivial system to assess their performance (Section 5).

4. Probe Distribution on Variants

This section characterizes the probe distribution problem, illustrates different distribution approaches, states the research questions, and provides algorithms to generate distributions with different properties.

4.1. Problem Characterization

Let us define U as the set of locations in program P that need to be instrumented to profile events E, and H as the set of probes actually inserted in U. To profile all events E, the engineer can generate an instrumented variant of P, v_0, with probes placed in locations $u_1, u_2, ..., u_k$ in P such that $H = U$ and $k = |H| = |U|$. This approach, however, may cause an unacceptable performance overhead. To address this problem, the engineer can specify an acceptable maximum number of probes h_{bound} that can be included in variant v_0 to enable the profiling activities. h_{bound} can be computed such that, for example, profiling overhead is hidden in the program's operating noise (performance variation observed during various program executions).

Since generally $h_{bound} << |U|$, data collected from all the sites when only *one* variant with h_{bound} probes is deployed will reflect only a part of the program behavior exercised in the field. Having multiple deployed variants of the program, where each variant has a somewhat complementary subset of assigned probes, can offer a better characterization of E [5]. This approach will result in n variants v_0, v_1, ..., and v_{n-1}, where each variant contains a subset of probes Hv_0, Hv_1, ..., and Hv_{n-1}, such that the size of each subset is less or equal to h_{bound}. We simplify the discussion by assuming that $|Hv_0| = |Hv_1| = ... = |Hv_{n-1}| = h_{bound}$.

Such a distribution of probes must consider at least two potential difficulties. First, **an event $e_i \in E$ may require probes in multiple locations in order to be profiled**. For example, to find out whether method A and method B always occur together (the event consists of their joined occurrence during a program execution), distributing probes such that the occurrence of A is captured in v_1 and the occurrence of B is captured in v_2 would not yield the wanted information. Second, **an engineer interested in events in E may find it easier and less expensive to approximate the locations in the program that must be profiled to capture those events**, than identifying all the exact locations where probes are needed. For example an engineer may utilize a call graph to quickly (over)estimate which methods may be involved in a call-chain. We refer to this set of program locations as a cluster of probes, c.

Once h_{bound}, n, and E (or the cluster of probes c to approximate E) are defined, the challenge is to find a distribution of probes across variants that maximizes the likelihood of capturing a representative part of the program behavior exercised in the field. More formally, we define the probe distribution across variants problem as follows:

Given:

U**,** a set of units in P, $u \in U$, and where u_i identifies a potential location for probe h_i;

E**,** a set of events of interest, $e \in E$, where e_i corresponds to a set of associated program locations required to capture e_i.

C**,** a set of clusters of probes in P, $c \in C$, where c_i consists of a set of probes that over-approximates the locations required to profile a set of target events in E.

PD**,** the set of potential probe distributions across C on n variants such that $\forall$ variants : $|Hv_0|$=$|Hv_1|$=...=$|Hv_{n-1}| = h_{bound}$ [1];

f**,** a function from PD to a real number that, when applied to any such distribution, yields an *award value*.

Problem: Find a distribution $D \in PD$ such that $\forall D' \in PD$, $D \neq D'$, $[f(D) \geq f(D')]$.

The definition of f will depend on the information that is targeted. For example, f may be the number of blocks covered, the potential invariants violated, or the number of call-sensitive-chains. Depending upon the choice of f, the problem of finding the best distribution may be intractable. Thus, in practice, we resort to heuristics to approximate D.

4.2. Potential Distribution Strategies

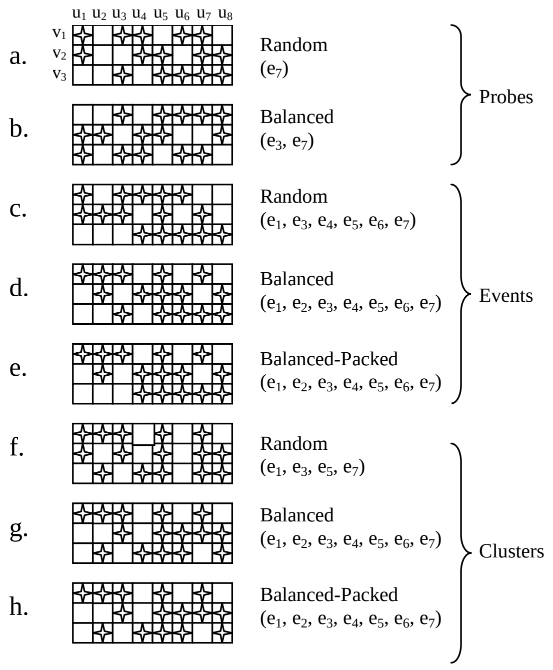
There are many potential strategies that can be applied in an attempt to maximize the award value of a probe distribution across variants. Figure 1 presents three types of strategies according to the distribution target: *Probe-based*, *Event-based*, and *Cluster-based*. We instantiate those types with sample distributions of increasing complexity to illustrate the range of potential solutions and their associated tradeoffs. In each case we assume that $|U| = 8$, E=$\{e_1$, e_2, e_3, e_4, e_5, e_6, $e_7\}$, $h_{bound} = 5$, and $n = 3$. We will also assume that $e_1 = \{u_1, u_3, u_5\}$, $e_2 = \{u_2, u_4, u_6\}$, $e_3 = \{u_3, u_5, u_7\}$, $e_4 = \{u_4, u_6, u_8\}$, $e_5 = \{u_1, u_2, u_3\}$, $e_6 = \{u_4, u_5, u_6\}$ and $e_7 = \{u_6, u_7, u_8\}$.

The first type of strategy, *Probe-based*, distributes probes across the program variants without explicit knowledge of the profiled events. This type of strategy works well for events that can be profiled with single probes (e.g., block coverage, method coverage). One way of distributing probes is by incrementally placing a probe into a random location in a variant until h_{bound} is achieved (*Random-Probes*). This strategy is simple and avoids an engineer's unintended bias to concentrate probes in certain sections of a program variant (e.g., variants with probes in rarely executed units will provide no information from the field). However, such a distribution can lead to an uneven distribution of probes. For example, in Figure 1-a, no probes are allocated to capture information relative to u_2, but all variants allocate one probe to profile the activity of u_7. The *Random-Probes* strategy may be improved by considering the notion of *balance*. Balancing attempts to generate a distribution with the same number of probes assigned to all program locations across variants. *Balanced-Probes* in Figure 1-b solves the inequities of *Random-Probes* by keeping track of the number of probes allocated per location across variants, distributing probes only in the locations with fewer probes.

[1]Note that a distribution across C subsumes one across E. Distributing probes on events is equivalent to doing so over a series of clusters where each cluster corresponds to just one event.

In the presence of events such as call-chains that require the instrumentation of multiple program locations, *Probe-based* type strategies are likely to miss and even provide false information. For example, the distribution generated with *Balanced-Probes* in Figure 1-b can only provide information about events e_3 and e_7 using v_1, while v_2 and v_3 fail to capture any event, and the execution of event e_1 under v_1 may lead us to infer that e_3 occurred simply because u_1 did not contain a probe in that variant. *Event-based* type distributions start addressing this problem by distributing sets of probes, one set per event. The first variation of this strategy, *Random-Events* (1-c), randomly chooses an event and places the probes associated with that event into a variant as long as it does not exceed h_{bound}. This strategy shares the same limitation as *Random-Probes*. For example, we note that the resulting distribution does not allocate probes to capture information relative to events e_2 but two variants allocate probes to profile the activity of events e_1 and e_6. The second variation of the *Event-based* strategy attempts to balance the events that receive probes across the variants. The strategy *Balanced-Events* keeps track of the events previously included and performs allocation considering only the events that are used less frequently. In Figure 1-d, probes are placed so that all the occurring events can be observed in at least one variant. A similar strategy, called *Balanced-Packed-Events* improves the previous strategy by trying to pack as many events as possible into each variant taking into consideration that some events share probes. In the program example, events e_4, e_6, and e_7 shared locations u_4, u_6, and u_8. By putting the probes needed to monitor these events in the same variant, we can fit more events into other variants potentially increasing our event coverage in the field. Figure 1-e shows that with this strategy, we are able to pack one more event in the third variant to monitor three events (e_4, e_6, and e_7) in contrast to only two events (e_3 and e_7) with the *Balanced-Events* strategy.

A similar type of strategy uses clusters of probes rather than individual events and is called *Cluster-based*. This type of distribution is particularly valuable when enumeration of the target events is unaffordable or cannot be done precisely. Probes that may be required to profile an event of interest are clustered together. For example, a cluster can be generated based on the examination of the structural characteristics of the program (e.g., form a cluster with all the probes in the same package or execution path) or by relying on an engineer's experience (e.g., form a cluster with all the probes exercised by each system test or not covered by the current suite). Following the previous example, suppose that C=$\{c_1, c_2, c_3, c_4, c_5, c_6\}$ where $c_1 = \{u_1, u_3, u_5, u_7\}$, $c_2 = \{u_1, u_2, u_3\}$, $c_3 = \{u_3, u_5, u_7, u_8\}$, $c_4 = \{u_4, u_5, u_6, u_8\}$, $c_5 = \{u_2, u_4, u_6\}$, and $c_6 = \{u_6, u_7, u_8\}$. Each of the clusters consist of a set of probes that is an over-approximation of an event (e.g., c_3 is an over-approximation of e_3), corresponds to a subset of events (e.g., c_1 over-approximates e_1 and e_3), or corresponds to exactly one event (e.g., c_2 captures e_5). Note that the three variations of *Event-based* (random, balanced, packed) distribution strategies can also be applied to *Cluster-based* since both types deal with the distribution of sets of probes, rather than individual ones.



u1 u2 u3 u4 u5 u6 u7 u8
v1
v2
v3
a.
Random (e7)
b.
Balanced (e3, e7)
Probes
c.
Random (e1, e3, e4, e5, e6, e7)
d.
Balanced (e1, e2, e3, e4, e5, e6, e7)
Events
e.
Balanced-Packed (e1, e2, e3, e4, e5, e6, e7)
f.
Random (e1, e3, e5, e7)
g.
Balanced (e1, e2, e3, e4, e5, e6, e7)
Clusters
h.
Balanced-Packed (e1, e2, e3, e4, e5, e6, e7)

Figure 1. Sample Distributions.

4.3. Distribution Algorithms

In Section 4.1 we defined a function $f(D)$ that produces an award value on a given distribution D. Finding the distribution that generates the highest award value can be formulated as an optimization problem [20]. An optimization problem minimizes or maximizes an objective function that evaluates whether an individual solution (in this case a single distribution) is better or worse than another one given a set of constraints. For a given probe distribution we define an objective function where zero represents the optimal distribution, with two constraints: there are a fixed number of variants, and all variants have exactly an h_{bound} number of probes. In this section we will define a cost function for each distribution, $cost(D)$, that when minimized provides the highest award value for $f(D)$. We then describe an algorithm for minimizing each of the $cost(D)$s defined.

In previous work we introduced a simple greedy algorithm to distribute probes for the *Balanced-Probes* scenario[3]. In this distribution it is always possible to obtain a range of one between the minimum and maximum number of times a probe is used across variants. Let q be the minimum number of times a given location has a probe inserted. Let $x_i = 1$ if the number of probes inserted in location i across all variants is greater than $q+1$, and 0 otherwise. Then $cost(D) = \sum_{i=1}^{location} x_i$. This function counts the number of locations that exceed the optimal range of probe use. When $cost(D) = 0$, the distribution is balanced.

Algorithm 1 Balanced Distribution

```
ALLOCATE PROBES($h_{bound}$, $minProbeLocationCount$){
$count$ = 0
$AvailableLocations$ = SelectLocations($minProbeLocationCount$)
while $count < h_{bound}$ do
    $s$ = selectRandomLocation($AvailableLocations$)
    place probe in location $s$
    remove location $s$ from $AvailableLocations$
    if $AvailableLocations$ is empty
        $minProbeLocationCount++$
        $AvailableLocations$ =
         SelectLocations($minProbeLocationCount$)
    $count++$
}
```

The *Balanced-Probe* distribution algorithm, (shown as Algorithm 1), places probes in random locations maintaining the balance at each step. The algorithm tracks the number of probes assigned to locations with the least number of probes ($minProbeLocationCount$). It starts by adding all locations to the $AvailableLocations$ set. Next, it randomly selects a location, s, from $AvailableLocations$, assigns a probe to s, and removes s from $AvailableLocations$. When $AvailableLocations$ is empty (the same number of probes have been assigned to all locations), $minProbe$-$LocationCount$ is incremented and $AvailableLocations$ is re-initialized. This process is repeated until h_{bound} probes are assigned.

The *Balanced* and *Balanced-Packed* distributions for events and clusters present a more complicated optimization problem. We can no longer easily calculate if a solution exists where all probes can be distributed given the h_{bound}, number of variants, and number of locations. With events of varying lengths and overlapping probes, it would require an exhaustive enumeration of all combinations of events onto the set of variants. Given the fact that there may be thousands of events this becomes combinatorially infeasible.

To solve the distribution problem for events and clusters we choose a different algorithmic strategy. We employ a standard optimization technique: a heuristic search. We use a hill climb to achieve balance and/or packing[20]. This algorithm will not guarantee that a global minimum is found for $cost(D)$, but may converge instead on a close to optimal local minimum. The hill climb algorithm (see Algorithm 2) starts with an initial random distribution of events. It then performs a series of transformations. At each stage it selects a random variant and a random number of events to remove from the variant, and it then randomly selects new events to add back until h_{bound} is met. If the new distribution is better ($cost(D)$ is closer to 0) the new distribution is accepted, otherwise it is not and the counter for bad moves is incremented. The algorithm terminates when a cost of zero is obtained or when it is "frozen"- it has performed a set number of bad consecutive moves (i.e. it has converged on a local minimum).

A key element of the algorithm is the calculation of the cost computation at each transition. There are three elements to consider when defining $cost(D)$: balance, overlap, and whether or not all events (or clusters) have been distributed. We define the overall objective function as follows: $cost(D) = \alpha \times$ BALANCE $+\beta\times$PACKING $+\gamma\times$ALLUSED. The three weights α, β, and γ are real numbers between 0 and 1 that can be adjusted based on our objective.

In our implementation, BALANCE $= \sum_j^{|E|} length(e_j) \times |TimesUsed(e_j) - Average(TimesUsed(E))|$. The length multiplier is meant to compensate for the smaller chance events with more probes have to fit into a given distribution. To calculate PACKING, let ov_i equal the count of overlapping probes in variant i and $OV = \sum_i^n ov_i$. Then $pack_i = 0$ if $Max(OV)$==ov_i, and $\frac{1}{Max(OV)-ov_i}$, otherwise, and PACKING=$\sum_i^n pack_i$. ALLUSED equals the sum of the length of events that are not included in the distribution. We use various data structures that keep track of events and probes as they are inserted and removed to make the implementation more efficient. With these additional structures, calculating ALLUSED can be performed in constant time. The cost of calculating PACKING requires $O(n)$ time where n is the number of variants deployed. Since n is relatively small this does not have a large impact on program efficiency. The expensive part of the objective function is BALANCE. Although Average($TimesUsed(E)$) can be re-calculated in constant time, by maintaining a global sum, re-calculating the distance from this average for all events requires $O(|E|)$ time. This clearly has the greatest impact on the scalability of our algorithm.

5. Empirical Study

Our overall objective is to learn how to best distribute a bounded number of probes into multiple deployed program variants to capture the maximum amount of field information. More specifically, we aim to assess a series of strategies that utilize different probe distribution mechanisms to capture call-chains. The following sections introduce the independent variables, the metrics, the hypotheses, the object of study, the design and implementation of the study, and the threats to validity that may affect our findings.

Algorithm 2 Hill Climb

```
ALLOCATE PROBES ($h_{bound}$) {
D = createInitialRandomDistribution()
$badcounter = 0$
$FROZEN = MAXBADMOVES$
while $cost(D) > 0$ && $badcounter < FROZEN$ do
    $r$ = selectRandomVariant
    $c$ = selectNumberofEventstoRemove($r$)
    removeEvents($c$)
    while not $h_{bound}$ do {
        $s$ = selectRandomLocation($AvailableLocations$)
        place event probes in variant
    }
    if($cost(D$ with changed $r$)< $cost(D)$)
        commit changes to variant $r$
    else
        $badcounter$++
}
```

5.1. Definitions and Setup

Call chains. In this study, we define a call-site sensitive call-chain as a traversal of the call graph from a root node to a leaf node, where edges in the graph represent method invocations, and are labeled by the caller-site. (From this point on we refer to the call-site sensitive call-chains simply as call-chains). We denote the main method of the program and all methods that represent event handlers as root methods. We denote a method as a leaf if there exists no call from that method to any other methods in the program. We do not consider loops or any back edges in the graph, and we exclude calls to external libraries.

Object Selection. We chose MyIE as our object of study. MyIE is a web browser that wraps the core of Internet Explorer to add features such as a configurable front end, mouse gesture capability, and URL aliasing. MyIE was particularly attractive because its source code is available, it introduces many of the challenges of other large systems (e.g., size, interaction with 3rd party components, complex event handling and highly configurable), it is similar but not identical to other web browsers, and it has a small user base that we can leverage for our study. The version of MyIE source code utilized, available for download at sourceforge.net, has approximately 41 KLOC, 64 classes, 878 methods, and 2793 blocks. For our study we consider 1197 unit probes that correspond to the call-site sensitive call-chains.

Object Preparation. To collect the field data, we instrumented MyIE source code to generate a block trace. During a user's execution, the block trace is recorded in a buffer. When the buffer is full, the information is compressed, packaged with a time and site ID stamp to maintain the confidentiality and privacy of the sender, and sent to the in-house repository if a connection is available or locally stored otherwise. An in-house server is responsible for accepting the package, parsing the information, and storing the data in the database.

Test Suite. We required a test suite to generate an initial assessment of the overhead and bounds. To obtain such a test suite, we generated a set of tests that exercised all the menu items in MyIE. After examining the coverage results, from the black box test cases we added test cases targeting blocks that had not yet been exercised. We automated a total of 243 test cases that yielded 79% of block coverage.

5.2. Independent Variables

Probes Placement Types and Techniques. We implemented the three types of techniques illustrated in Section 4.3:

- Probe-based distributions: Random-Probes (R_Pr) and Balanced-Probes (B_Pr)
- Event-based distributions: Random-Events (R_Ev), Balanced-Events(B_Ev), Packed-Events(P_Ev) and Balanced-Packed-Events(BP_Ev).
- Cluster-based distributions: Random-Clusters (R_Cl), Balanced-Clusters(B_Cl), Packed-Clusters (P_Cl), and Balanced-Packed-Clusters(BP_Cl)

While the probe-based distribution techniques operate without prior-information about the events to profile, the event-based and the cluster-based techniques require some initial call-chain information.

To apply the event-based distribution techniques we needed to generate an initial list of events (call-chains). We generated this statically by analyzing a call-graph of the application with the support of the Microsoft Studio C++ 6.0 navigation tool-set. We hand-annotated the edges with the caller-site, adding extra edges when a caller invoked a callee from multiple locations. We validated the graphs by examination and by running an available test suite to detect any other potential edges missed by the static analysis tool. We then generated the list of call-chains in our object of study by performing a depth-first search traversal of the graph. [2]

To apply the cluster-based techniques, we did not need to identify apriori the set of events to profile. Instead, we had to use a heuristic to define the clusters of probes that may contain the events of interest. We again utilized the call-graph to identify our clusters, where each cluster included one root note and all the methods reachable from that root.

For balancing and packing the event-based and cluster-based distributions, we utilized Algorithm 2, with the parameters presented in Table 1.

[2]Although we are aware of more precise techniques for generating call-chains (we later discuss why we did not pursue them and the impact of such a choice), it is important to recognize that this is an inherent limitation of this type of technique.

Table 1. Simulation Parameters

Technique	α	β	γ	FROZEN
Balanced	0.5	0	0.5	500,000
Packed	0	0.5	0.5	500,000
Balanced-Packed	0.4	0.2	0.4	500,000

Bounds. We defined four levels of overhead: 5%, 10%, 25%, and 50% over the non-instrumented program. To approximate the number of probes corresponding to the chosen levels of overhead, we inserted a number of probes in random locations of the program associated with call-chains, ran an available test suite, measured the overhead, and repeated the procedure while adjusting the number of inserted probes until we converged at the target overhead level. This resulted in four bound levels (h_{bound}) as defined by the following number of probes: 50, 75, 230, and 450.

5.3. Dependent Variables

The dependent variable is the captured field information value, for which we have selected two metrics. The first metric is the percentage of call-chains covered in the field when using a given probe allocation technique with respect to the call-chain coverage obtained by a theoretical optimal allocation technique, Opt. Given an h_{bound} level and the data collected from the field, Opt represents what would have been the ideal distribution of probes across variants. Since we captured a complete block trace during the deployment of $MyIE$, we were able to approximate the Opt distribution aposteriori by removing the probes corresponding to call-chains that were observed in multiple variants, or by removing the shortest call-chain on a variant.

The second metric aims to measure the false call-chains reported by each technique due to missing probes. A call-chain detected in the field is considered false if (1) the call-chain is not detected by the $Full$ technique (technique that inserts instrumentation in all units of the program) or (2) the call-chain is detected by the $Full$ technique, but it is not detected at the same location in the trace file as $Full$.

5.4. Hypotheses

We are interested in obtaining the degree to which the amount of information collected from the field changes across the probe distribution strategies when h_{bound} is varied. We formally state the primary null hypotheses (assumed to be true until statistically rejected) in Table 2.

5.5. Deployments and Data Collection

We first performed a set of preliminary deployments of $MyIE$ within our lab to verify the correctness of the installation scripts, data capture process, magnitude and frequency of data transfer, and the transparency of the de-

[3]As defined by the dependent variables in Section 5.2.

Table 2. Hypotheses

Null Hypotheses	There is **no significant ...**
$H1$	performance[3]difference between probe-based (Pr), event-based (Ev), and cluster-based techniques (Cl) of the same type.
$H2$	performance difference between Random (R), Balanced (B), Packed (P), and Balanced-Packed (BP) techniques.
$H3$	difference when using different h_{bounds}.
$H4$	interaction between types, techniques, and h_{bounds}.

installation process. After this initial refinement period, we proceeded to perform a full deployment and started with the data collection. We sent e-mail to the members of our Department and various testing newsgroups (e.g. comp.software.testing, comp.edu, comp) inviting them to participate in the study and pointing them to our MyIE deployment web site for more information. After 3 months, there were 114 downloads, and 36 deployed sites that qualified for this study, which generated 378 user sessions.

We utilized the collected data to simulate each one of the combinations of distribution techniques and types. Each simulation generated n variants, where each variant consisted of a vector of size equal to the number of locations in the program. Cells in each vector were initialized with zeroes, and then populated with h_{bound} probes according to the allocation rules specified by the simulated technique. Each vector was then utilized to mask the data collected from a specific deployed instance, simulating what would have been collected if a true variant would have been deployed to that particular instance. We performed the variant generation and assignment process ten times to account for potential variations due to the random assignment of probes to variants, and from variants to instances. For this study, we assume that the number of variants $n = 36$, that is, we have as many variants as deployed instances which constitutes an upper bound on the potential distribution.

5.6. Threats to Validity

From a generalization perspective, our findings are limited by the object of study, the data-collection process, and the user population. Although it is arguable whether the selected program is representative of the population of all programs, there are many similar browsers to MyIE, making it a credible experimental object. During instrumentation and data collection we attempted to balance data representativeness and power through the utilization of full data capture combined with simulation. The deployment and download process, as perceived by the users, was identical to many other sites offering software downloads and on-line patches. Further studies with other programs and subjects may be necessary to confirm our findings.

From an internal validity perspective, the quality of the collected field data is an important threat as packages may be lost, corrupted, or not sent. We controlled this threat by adding different stamps to the packages to ensure their authenticity and to detect any anomalies in the collection process. Alternative definitions (e.g., fixed-length call-chains, component-bounded call-chains) may influence the number detected or the noise. The static analysis to generate the initial list of call-chains for the event-based technique may have limited their performance as well. Our choice of definition was driven by practical considerations including limitations in the static analysis tools which cannot fully process large C++ applications that utilize MFC components.

From a construct validity perspective, we have chosen a set of metrics to quantify the value of the collected information that captures only a part of its potential meaning. Our choices are a function of our interest in exploiting field data for validation purposes, and our experience and available infrastructure to analyze such data. Also, from a construct perspective, we have approximated the h_{bound} and have chosen a subset of the potential levels for it that attempt to operationalize a spectrum of values that allow us to characterize the effects of the treatments.

From a conclusion validity perspective, we are making inferences based on a few hundred sessions which may have limited the power to detect significant differences. However, we were able to reject various hypotheses and discover interesting findings.

6. Results and Analysis

We now compare the performance of probe distribution types (Probe-based, Event-based, and Cluster-based) and techniques (random, balanced, and packed) across different bound levels that restrict the number of probes placed on a given variant. We start by performing an exploratory analysis of the data through the box plots in Figure 2, which depicts the percentage of call-chains correctly identified by the deployed instances (compared to what is achieved by the *Opt* technique) when utilizing the distribution techniques for the chosen four levels of h_{bound} (one per subfigure).

Event-based and Cluster-based techniques (the ones in the gray area of Figure 2 and to their right) perform better than Probe-based techniques in terms of identifying correct call-chains for all bound levels (the two left most boxes of each subfigure). This suggests that simply allocating probes without associating them with the target events is unlikely to provide much valuable data even in the presence of h_{bound} values as high as 50%. We also note that Event-based techniques perform at least as well as Cluster-based techniques

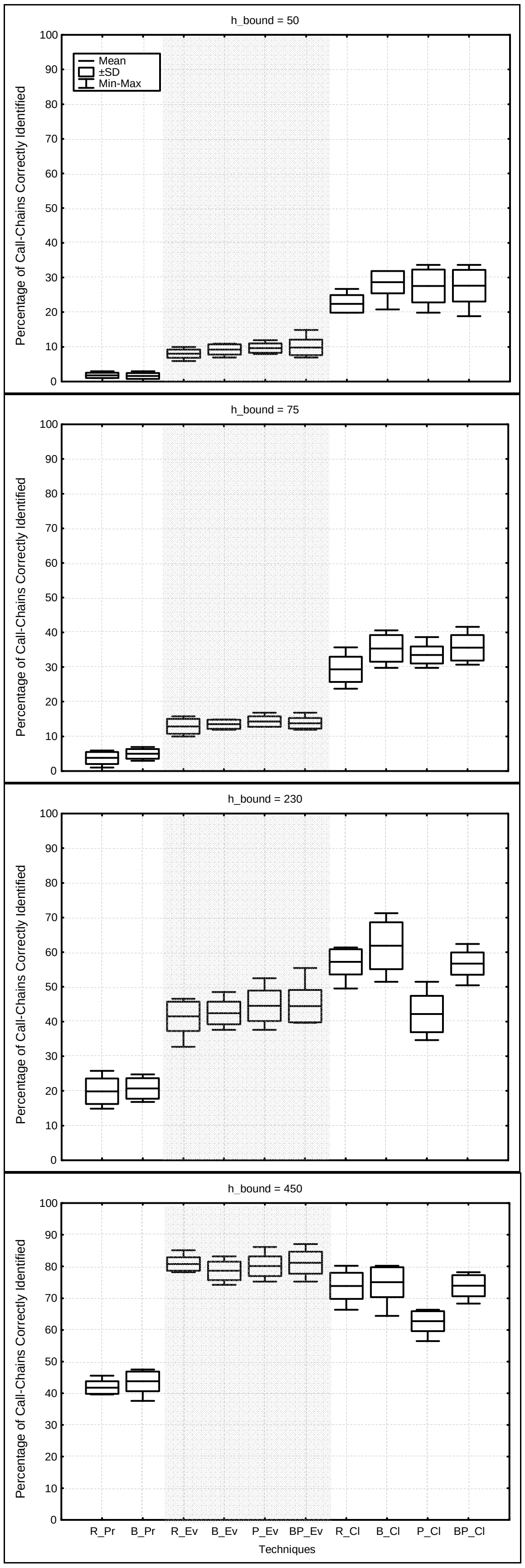


Figure 2. Identification of Call-Chains

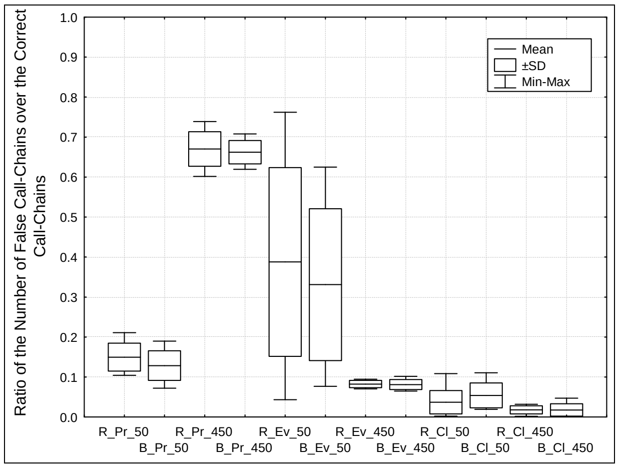


Figure 3. Falsely Reported Call-Chains

when h_{bound} = 450, but consistently worse for lower h_{bound} values. We conjecture that the poor performance for lower h_{bound} might be caused by the imprecision in determining individual events apriori. *This confirms that Event-based techniques can be beneficial when applied to events that can be easily enumerated, but in general a Cluster-based approach seems to be better, independent of the h_{bound} constraints.* An engineer can take this into consideration when deciding which techniques to use.

When comparing random techniques to the search-based techniques (balanced, packed, and balanced packed) in terms of the correctly identified call-chains we found that, within Event-based techniques, the improvement provided is negligible, and within Cluster-based, balancing and packing may provide some benefits (6% over random on average).

To formally determine whether the observed tendencies were the result of chance and to evaluate our hypotheses, we performed an ANOVA on the dependent variable (correct call-chain identified) and the three independent variables (type of distributions, techniques and h_{bound}). The analysis over techniques and h_{bound} was performed independently within each distribution type. The summary of the p-values of the analysis is shown in Table 3. P-values smaller than 0.05 indicate a significant relationship between the treatment and the dependent variable that cannot be attributable to luck, and should be interpreted as a rejection of the null hypothesis.

The p-values in Table 3 show that the correct percentage of call-chains reported by the deployed sites varies significantly across different types of distribution and bounds ($H1$). Balancing did not seem to matter when utilizing Probe-based distributions, but it did within the Event-based and Cluster-based distributions ($H2$). As expected and overall, the h_{bound} level did affect the dependent variable ($H3$), but it was interesting to see that its effect within the Cluster-based distribution had a significant interaction

Table 3. p-values of the ANOVA Test

Hypo-theses	Effect		p-value
$H1$	Type(Probes, Events, Clusters)		*0.00*
$H2$	Techniques (R, B, P)	Probes	0.69
		Events	*0.02*
		Clusters	*0.00*
$H3$	h_{bound}		*0.00*
$H4$	Techniques & h_{bound} (R, B, P)	Probes	0.51
		Events	0.54
		Clusters	*0.00*

($H4$). We further explored this interaction through a Bonferroni analysis to quantify when a technique and bound level had an effect on the call-chain detection. We found that only when h_{bound} is 50 and 75, balanced and balanced-packed enabled the detection of significantly higher percentage of correctly identified call-chains than random.

Figure 3 shows the ratio of falsely reported call-chains over the correctly identified ones across the three types of distributions. Due to space constraints, for each type of distribution, we only show Random and Balanced techniques on two extreme bound values (50 and 450). We can see different tendencies between Probe-based, Event-based, and Cluster-based types as h_{bound} increases: the number of falsely reported call-chains by the Probe-based techniques, as shown by the 4 leftmost box-plots, increases while the number decreases for Event-based and Cluster-based types. We conjecture that in the case of Probe-based techniques, the more probes inserted leads to more incomplete events in a variant. In contrast to Event-based and Cluster-based types, increasing h_bound increases the likelihood that longer call-chains are accommodated. Furthermore, in addition to having a large variance (0.055), the number of falsely reported call-chains in Event-based techniques is 10 times higher than the Cluster-based technique when h_{bound} is 50. This value improves significantly when h_{bound} is 450, though it is still 8 times higher than the number falsely reported by the Cluster-based technique.

Implications. *When the acceptable overhead that bounds the profiling effort is small (less than 10% in our study), Cluster-based allocation techniques that balance the distribution across clusters of probes are the most effective in retaining the value of field data. When many probes can be placed in a variant, the Event-based type distribution seems to perform well regardless of the technique. In general, any distribution technique working with clusters reported significantly less false call-chains than the Probe-based and Event-based distributions, which translates into savings for the engineer investigating such chains.*

7. Conclusion

Profiling overhead limits what we can observe and learn from deployed software. To address this limitation, this paper investigates ways to distribute probes across variants to meet profiling overhead constraints while maximizing captured field information. We have formalized the problem of distributing probes across variants, presented several distribution techniques, and carefully assessed their performance.

From our findings we can draw several interesting observations. First, probe distribution techniques are consistently relevant when overhead bounds are set below 10% (a reasonable practice when the objective is to remain below the threshold of user noticeability). Second, distributions that balance probes across variants perform consistently better than those that do not, independent of the overhead bounds and the type of distribution. Last, Cluster-based distributions tend to be less expensive to set up than Event-based, collect more correct information than Event-based or Probe-based, and report the least false information.

As continuing work we are starting to address the various costs involved in large scale profiling efforts. For example, analyzing the target program to relate probes to events and to determine an optimal probe placement may be expensive. We have shown that approximations through clustering can be effective to lower those costs but we have not yet quantified this. The cost of the probe distribution algorithm is computationally expensive, taking days to converge on a distribution in the presence of thousands of target events and profiling locations. More flexible convergence criteria and more efficient implementations of the algorithms are necessary to make this approach scalable. Finally, we must weigh the additional costs introduced by deploying and maintaining multiple program variants, against the benefit of having more variants which lowers the amount of instrumentation per deployed instance. Future studies should examine the trade-off between variants and bounds, and more generally, the costs and benefits of large scale profiling efforts.

8. Acknowledgments

This work was supported in part by NSF CAREER Award 0347518 and an NSF EPSCoR First Award. We are thankful to Zhimin Wang for preparing MyIE, and the study's participants.

¡¡¡¡¡¡¡ paper.bbl

References

[1] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Conf. on Prog. Lang. Design and Impl.*, pages 168–179, 2001.

[2] T. Ball and J. Laurus. Optimally profiling and tracing programs. In *Symp. on Principles of Prog. Lang.*, pages 59–70, Aug. 1992.

[3] M. Diep, S. Elbaum, and M. Cohen. Profiling Deployed Software: Strategic Probe Placement. Technical Report TR-05-08-01, University of Nebraska - Lincoln, Aug. 2005.

[4] M. Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. In *Int. Workshop on Soft. and Performance*, pages 139–150, 2004.

[5] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Soft. Eng.*, 31(4):312–327, 2005.

[6] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Int. Symp. on Soft. Testing and Analysis*, pages 65 – 75, June 2004.

[7] A. Glenn, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.

[8] S. Graham and M. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN SCC*, 17(6):120–126, June 1982.

[9] M. Harrold, R. Lipton, and A. Orso. Gamma: Continuous evolution of software after deployment. www.cc.gatech.edu/aristotle/Research/Projects/gamma.html.

[10] D. Hilbert and D. Redmiles. An approach to large-scale collection of application usage data over the Internet. In *Int. Conf. on Soft. Eng.*, pages 136–145, 1998.

[11] InCert. Rapid failure recovery to eliminate application downtime. www.incert.com, June 2001.

[12] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 412–421, New York, NY, USA, 2005. ACM Press.

[13] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conf. on Prog. Lang. Design and Impl.*, pages 141–154. ACM, June 2003.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Conf. Prog. Lang. Design and Impl.*, pages 15–26, June 2005.

[15] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Int. Conf. on Soft. Eng.*, pages 449–458, May 2004.

[16] Microsoft. Windows quality online services, 2005.

[17] A. Orso, T. Apiwattanapong, and M.J.Harrold. Leveraging field data for impact analysis and regression testing. In *Foundations of Soft. Eng.*, pages 128–137. ACM, September 2003.

[18] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Int. Symp. on Soft. Testing and Analysis*, pages 65–69, 2002.

[19] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Int. Conf. of Soft. Eng.*, pages 277–284, May 1999.

[20] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith. *Modern Heuristic Search Methods*. John Wiley & Sons, Ltd., West Sussex, 1996.

[21] S. Reiss and M. Renieris. Encoding Program Executions. In *Int. Conf. of Soft. Engineering*, pages 221–230, May 2001.

[22] D. Richardson, L. Clarke, L. Osterweil, and M. Young. Perpetual testing. www.ics.uci.edu/ djr/edcs/PerpTest.html.

[23] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in java. *SIGSOFT Softw. Eng. Notes*, 29(4):1–11, 2004.

[24] A. L. Souter and L. L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, 44(13):721–732, 2002.

[25] M. Tikir and J. Hollingsworth. Efficient instrumentation for code coverage testing. In *Int. Symp. on Soft. Testing and Analysis*, pages 86–96, 2002.

=======

References

[1] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Conf. on Prog. Lang. Design and Impl.*, pages 168–179, 2001.

[2] T. Ball and J. Laurus. Optimally profiling and tracing programs. In *Symp. on Principles of Prog. Lang.*, pages 59–70, Aug. 1992.

[3] M. Diep, S. Elbaum, and M. Cohen. Profiling Deployed Software: Strategic Probe Placement. Technical Report TR-05-08-01, University of Nebraska - Lincoln, Aug. 2005.

[4] M. Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. In *Int. Workshop on Soft. and Performance*, pages 139–150, 2004.

[5] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Soft. Eng.*, 31(4):312–327, 2005.

[6] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Int. Symp. on Soft. Testing and Analysis*, pages 65 – 75, June 2004.

[7] A. Glenn, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.

[8] S. Graham and M. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN SCC*, 17(6):120–126, June 1982.

[9] D. Hilbert and D. Redmiles. An approach to large-scale collection of application usage data over the Internet. In *Int. Conf. on Soft. Eng.*, pages 136–145, 1998.

[10] InCert. Rapid failure recovery to eliminate application downtime. www.incert.com, June 2001.

[11] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 412–421, New York, NY, USA, 2005. ACM Press.

[12] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conf. on Prog. Lang. Design and Impl.*, pages 141–154. ACM, June 2003.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Conf. Prog. Lang. Design and Impl.*, pages 15–26, June 2005.

[14] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Int. Conf. on Soft. Eng.*, pages 449–458, May 2004.

[15] Microsoft. Windows quality online services, 2005.

[16] A. Orso, T. Apiwattanapong, and M.J.Harrold. Leveraging field data for impact analysis and regression testing. In *Foundations of Soft. Eng.*, pages 128–137. ACM, September 2003.

[17] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Int. Symp. on Soft. Testing and Analysis*, pages 65–69, 2002.

[18] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Int. Conf. of Soft. Eng.*, pages 277–284, May 1999.

[19] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith. *Modern Heuristic Search Methods*. John Wiley & Sons, Ltd., West Sussex, 1996.

[20] S. Reiss and M. Renieris. Encoding Program Executions. In *Int. Conf. of Soft. Engineering*, pages 221–230, May 2001.

[21] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in java. *SIGSOFT Softw. Eng. Notes*, 29(4):1–11, 2004.

[22] A. L. Souter and L. L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, 44(13):721–732, 2002.

[23] M. Tikir and J. Hollingsworth. Efficient instrumentation for code coverage testing. In *Int. Symp. on Soft. Testing and Analysis*, pages 86–96, 2002.

¿¿¿¿¿¿¿ 1.2