2010

Monitoring software using property-aware program sampling

Harish Narayanappa
Iowa State University

Monitoring software using property-aware program sampling

by

Harish B Narayanappa

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
David Fernández-Baca
Andrew Miner

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I take this opportunity to thank Dr. Hridesh Rajan for his guidance and support during my research on this work. His research advice and emphasis on collaboration has been of great influence on my thesis work and my graduate studies at the university. He has also helped me improve my communication, specifically the technical writing skills. Additionaly, in the beginning, he was patient with me when I was struggling to find a thesis problem to work on.

I am thankful to Dr. Mukul S. Bansal for his collaboration on formalizing our ideas to make it more precise and complete. He has been forthcoming and readily available for the discussions on any issues or questions we faced.

I would also like to thank my lab mates, specifically Youssef Hanna and Rob Dyer, for their review and discussions on the various approaches proposed. I would also like to acknowledge Mahantesh Hosamani, as our collaborative work provided me the thesis problem to work on.

I am grateful to my parents B.R.Narayanappa and H.B.Shylaja, for their encouragement and moral support during my time on this thesis work.

ABSTRACT

Monitoring and profiling programs provides us with opportunities for its further analysis and improvement. Typically, monitoring or profiling, entails program to be instrumented to execute additional code that collects the necessary data. However, a widely-understood problem with this approach is that program instrumentation can result in significant execution overhead. A number of techniques based on statistical sampling have been proposed to reduce this overhead. Statistical sampling based instrumentation techniques, although effective in reducing the overall overhead, often lead to poor coverage or incomplete results. The contribution of this work is a profiling technique that we call *property-aware program sampling*. Our sampling technique uses program slicing to reduce the scope of instrumentation and *slice fragments* to decompose large program slices into more manageable, logically related parts for instrumentation, thereby improving the scalability of monitoring and profiling techniques. The technical underpinnings of our work include the notion of slice fragments and an efficient technique for computing a reduced set of slice fragments.

CHAPTER 1. INTRODUCTION

An insight into the runtime behavior of a deployed software application provides potential opportunities for its improvement. This has become more relevant today as the software evolves to become more complex with shorter development and release cycles and limitations of simple static analysis and testing techniques. Profiling or monitoring software provides such insights. The dynamic information gathered from monitoring and profiling can be used in performance tuning of applications [12], coverage-based testing [24], analyzing the application's usability etc.

For monitoring or profiling and for tasks involving runtime analyses such as bug detection [27, 20], continuous testing [2], dynamic optimization [5], it is often necessary to instrument programs. Full instrumentation to collect data about all interesting program points is reported to cause between 10% and 390% time and space overhead [14, 27]. A number of techniques have been proposed to reduce the instrumentation overhead. Statistical sampling-based techniques, for example, instrument a randomly selected, relatively smaller, subset of program points [4, 27]. A problem with random sampling of program points is that it often provides inadequate profile of program points relevant to properties of interest [20]. On the other hand if the selection of program points was based on particular concern or criteria, intuitively, the generated profiles are likely to be more coherent in understanding the software better. A problem with random sampling of program points is that to obtain an adequate profile of program points relevant to a property of interest requires an unnecessarily large amount of samples [20]. Among other things, a profile with respect to a property of interest is helpful in focusing the developer's attention, and for reducing the complexity of profiling results [35].

To keep the cost of instrumentation low, while maintaining increased coverage with better monitoring information and runtime profiles, we contribute the notion of *property-aware program sampling*. A key insight, which we borrow from Hatcliffet al. [19] among others, is to focus the instrumentation

efforts on parts of program that are relevant for a property of interest, i.e. the program slice, with the property as a slicing criterion [38, 32]. This helps focus instrumentation efforts on the desired parts of a program. The computed program slices may tend to be as large as the program itself, especially in well-modularized programs, where the modules are largely *cohesive* in nature [39]. To resolve this, our profiling technique randomly samples statements from the program slice, as compared to sampling from all statements in the program as in earlier approaches [4, 27].

An important issue to consider here is that randomly sampling statements provides inadequate coverage of the implicit control relations between sampled statements. These implicit path profiles have potential applications in performance tuning, continuous program optimizations, software test coverage, hot-path prediction and profile-directed compilation with respect to the property under consideration. A random sample of statements may eventually cover all elements along different paths, thus giving the needed profile information, however, such coverage would be infrequent. Moreover, if such coverage is essential more samples would be needed, which would increase instrumentation overhead.

To address this issue, we introduce another strategy based on sampling a population that consists of *slice fragments*. Informally, a *slice fragment* consists of a subset of the statements in the program slices (we provide formal definitions in Section 2). The statements in a slice fragment are logically related in that they capture the implicit control structure between the statements of the program slice. During sampling, whenever a slice fragment is selected, there is a higher probability that the profile data for all constituent statements will be collected. This translates to higher probability of better profile with lower instrumentation overhead.

Our sampling strategy based on slice fragments is beneficial in cases where the slice of a program may itself become very large [37, 39]; in some cases as large as the program. It provides a tradeoff between overhead and profile information. It has lesser overhead compared to full program profiling. It has more overhead compared to random sampling of program statements, but provides much more information about implicit control paths, with potential benefits for use cases such as feedback-based optimization.

1.1 Thesis Outline

The rest of this thesis is presented in the following chapters, in order, as follows:

Background: This chapter provides the background necessary for further chapters. It provides explanation of program slicing with the help of examples. Then briefly discusses concept of random sampling. It also provides formal definitions of important concepts upon which we develop our approach.

Approach: We restate the main problem and arising sub-problems addressed in this thesis. Further in the chapter we discuss the steps of our approach - *property-aware program slicing* in detail.

Evaluation: We evaluate the various properties of our approach in this chapter. The details of the tools and environment, followed by the experimental results such as, statistical significance and coverage properties of random sampling on fragments are discussed here. Lastly, we provide sample runtime analysis for a few programs under various sample sizes.

Related Work: In this chapter, we briefly list other studies which share similar goals and have provided us with useful insight. The differences of our approach with respect the stated related works are also mentioned.

Conclusion: We introspect on the contribution of this work with a summary. The limitations of the work in its current state and possible enhancements for the future are also mentioned

CHAPTER 2. BACKGROUND

2.1 Program Slicing

Program slicing is a technique for computing a subset of a program with respect to a property of interest. The result of this slicing is a program slice, which contains parts of the program which have influenced, either directly or indirectly the property of interest, which is stated as a slicing criterion. Weiser [38] first proposed this concept in the context of debugging. Since then, several notions of program slicing have been proposed with applications in software testing, program comprehension, software maintenance, software measurement, reverse engineering among others. For surveys on program slicing, refer Tip [37] and Bao [39].

To compute a program slice, one needs to specify a slicing criterion in the beginning. It is usually stated as tuple (p, $\mathtt{V}$), where p identifies program point where the computation is to begin and $\mathtt{V}$ specifies the subset of variables in the program reflecting the property of interest. The result of program slicing based on a slicing criterion $C = (p, \mathtt{V})$, is a decomposed subset which contains only those statements that are likely to influence the values of variable stated in the slicing criterion.

2.1.1 Example: backward slicing

To illustrate the above with an example lets consider an example program in Figure 2.1, which processes an input order $Order$. Suppose we specify the slicing criterion as <41, $discount$> for program slice computation, where the program point here is specified by the line number of the statement in the program and $discount$ is the program variable of interest, then the shaded regions in the Figure 2.1 represent the program slice - statements which influence the value of $discount$ at statement present at line number 41.

The slicing illustrated so far represents *backward* slicing - subset of program which directly or

```
1 public int processOrder(Order order) {
          ...
3         int discount = 0;
          ...
5         int userName = order.userName;
6         int userId = getCustomerId(userName);
7         int itemId = order.itemId;
          if (! validateCustomer(userId) ) {
                  ...
10                logInvalidOrder(userId, itemId);
11                return ERROR_CODE;
12        }
          ...
14        if (paymentMode == CREDIT_CARD) {

                  if (! creditCardValid(order.card) ) {
                          ...
18                        logInvalidOrder(userId, itemId);
19                        return ERROR_CODE;
20                }

22                if (isPremiumMember(userId))
23                        discount += 0.05;

25                if (overStockedItem(itemId))
26                        discount += 0.03;
                  ...
28                updateInvetory(itemId);
                  ...
30                recordCCTransaction(userId, itemId, price);
31                return SUCCESS;
32        }
          ...
34        if ( isPremiumMember(userId) ) {
35                discount += 0.03;
36        }

38        price = price - price * discount;
          ...
40        updateInvetory(itemId);
          ...
42        recordTransaction(userId, itemId, price);
          ...
44        logPerformance(...);
          ...
46        return SUCCESS;
47 }
```

Figure 2.1 An example of a backward slicing: shaded regions represent the backward slice with <41, $discount$> as the slicing criterion

indirectly *affects* the value of a given variable at a particular statement. This was the original notion of program slicing which helped debug unexpected values of variables at program statements by tracing back in the program slice. In the context of debugging, program slices help reduce the scope of the task and hence better productivity.

2.1.2 Example: forward slicing

A forward program slice consists of all statements in the program which are *affected by* the variables in the slicing criterion [32, 34]. Taking the same example program as before, we illustrate the forward slice with respect to slice criterion $C = (5, userName)$. In Figure 2.2, the shaded statements represent the forward program slice. The computation of forward slices is similar to that of backward slices, except that forward dependencies are taken into account. Forward slices are applicable to scenarios program comprehension and software re-engineering, where they help reduce the scope of task.

Program slicing have various other applications which are similar in treatment to the examples shown previously. These various applications make use different properties of program slices.

2.2 Random Sampling

The goal of statistical sampling is to estimate or infer some property of the entire population based on the data collected from only a subset of elements - *samples*. Assuming that the size of the population is N, let s_1, s_2,...s_N represent the constituent elements of the population . A subset of this population is said to be a random sample if the subset s_i, s_{i+1},...s_{i+n} containing n elements is chosen at random from all $\binom{N}{n}$ distinct possible subsets in which no element is included more than once. In other words, each of the $\binom{N}{n}$ subsets has the same probability $\binom{N}{n}^{-1}$ of being chosen. A simple random sample of size n can be obtained by randomly drawing elements from the population one at a time, n times, without replacing a drawn element back in the population i.e. once an element has been chosen in a sample, it is removed from subsequent drawings [7, 6].

An important factor to consider during sampling is choosing an appropriate sample size which has directly proportional to the sampling cost. The sampling size is correlated with the accuracy and *confidence* of the *statistical inference*. If the sample size is small, then more samples need be taken for

```
1 public int processOrder(Order order) {
        ...
3       int discount = 0;
        ...
5       int userName = order.userName;
6       int userId = getCustomerId(userName);
7       int itemId = order.itemId;
8       if (! validateCustomer(userId) ) {
                ...
10              logInvalidOrder(userId, itemId);
11              return ERROR_CODE;
12      }
        ...
14      if (paymentMode == CREDIT_CARD) {

16              if (! creditCardValid(order.card) ) {
                        ...
18                      logInvalidOrder(userId, itemId);
19                      return ERROR_CODE;
20              }

22              if (isPremiumMember(userId))
23                      discount += 0.05;

25              if (overStockedItem(itemId))
26                      discount += 0.03;
                ...
28              updateInventory(itemId);
                ...
30              recordCCTransaction(userId, itemId, price);
31              return SUCCESS;
32      }
        ...
34      if ( isPremiumMember(userId) ) {
35              discount += 0.03;
36      }

38      price = price - price * discount;
        ...
40      updateInventory(itemId);
        ...
42      recordTransaction(userId, itemId, price);
        ...
44      logPerformance(...);
        ...
46      return SUCCESS;
47 }
```

Figure 2.2 An example of a forward slicing: shaded regions represent forward slice with <5, $userName$> as the slicing criterion

better analysis, while on the other hand if the sample size is too large, the benefit using of statistical sampling in first place may be diminished. There is always a tradeoff between accuracy and *confidence* of the *statistical inference* against the sampling cost: the better the confidence, the more expensive the sampling.

Random sampling is typically applied to situations, where the generation of the population is not predictable and its elements have no pre-defined behavior. Analysis of one or few random samples may depict incorrect or random behavior with respect to the whole population, but if sufficient samples are taken then results obtained are likely to be close to that of the population.

2.3 Basic Definitions

Let $G = (V, E)$ be a directed graph, where V is the set of vertices of the graph and $E \subseteq (V \times V)$ is the set of edges. Given vertices $v, v' \in V$, a path in G from v to v', denoted by $v \rightarrow^+ v'$, is defined as follows: $v \rightarrow^+ v' \Rightarrow (v, v') \in E$ or $\exists v_1, \ldots, v_k \in V$ such that $\{(v, v_1), (v_1, v_2), \ldots, (v_k, v')\} \subseteq E$.

Definition 2.3.1 *A* **control flow graph** *is a directed graph $G_{cfg} = (V, E, v_0)$, where V is a set of nodes, representing a statement or group of statements, $E \subseteq (V \times V)$ is the directed edge set of the graph representing potential flow of execution between the nodes, and $v_0 \in V$ denotes a unique entry vertex. For convenience, it is assumed that all $v \in V$ are reachable from v_0 i.e. $\forall v \in V$, $v_0 \rightarrow^+ v$ holds.*

Definition 2.3.2 *A* **forward static slice** *$\mathcal{S}$ constructed from program, p with respect to crieria $\mathcal{C} = (X, c)$, is the set of statements and predicates that are affected by the values of any variables in X starting at program point c.*

Definition 2.3.3 *A* **back edge** *in a control flow graph $G_{cfg} = (V, E, v_0)$ is any edge $e \in E$ that points to an ancestor in depth-first(DFS) traversal of the graph.*

Any further references to "slice" or "program slice" in this thesis refers to a *forward static slice* [33, 32]. The **back edges** in **control flow graph** are encountered in case of loops, and recursion.

CHAPTER 3. APPROACH

In this chapter, we first restate the main problem along with the sub-problems we have attempted to solve. Further sections, discuss the *property-aware program sampling* in detail.

3.1 Problem restated

- Monitoring or profiling programs entails program instrumentation which gathers the necessary data at runtime. This instrumentation typically causes substantial execution overhead resulting in either incomplete results or avoidance of instrumentation in the program.

- Program slicing is the process of computing subset of statements in a program that directly or indirectly contribute to the computation performed at a given program point. In our approach, we propose application of program slicing to reduce the scope of monitoring and profiling. A well know problem with program slicing is that size of the slice may itself become large; in some cases as large as the program itself [39, 37]. This problem may render application of program slicing less lucrative.

- To manage the size of the program slice for instrumentation we decompose it into *slice fragments* - path or data related subset of program points in the program slice. This decomposition may lead to large number of *slice fragments*, plus some of them might be subsumed within one another. This raises the issue of fairness of selection during random sampling

3.2 Property-Aware Program Sampling

There are two key insights which drive our approach. Firstly, selecting a subset of program entities for monitoring and profiling is likely to reduce instrumentation overhead and facilitate efficient moni-

toring of a software application [3]. Program slicing, a static analysis technique, produces this subset of program entities that are relevant to a slicing criterion. Others have used this insight for reducing the scope of their task. For example, Hatcliff et al. [19] use program slice for reducing the size of the model that Bandera [10], a model checker for Java verifies. Guo et al. [18] use similar technique for limiting the input to their shape analysis technique, etc. We have used this technique to guide instrumentation for monitoring and profiling programs. Limiting the scope of the monitoring technique to the program slice may help achieve better profile of the parts of the program pertaining to the slicing criterion.

The second insight is that a slice need not be the unit of instrumentation as it often has the tendency to become large [37, 39]. Instead, only a part of it can be instrumented at a time. The instrumented part may vary guided by a statistical sampling plan. If the sample population is sufficiently large, samples are taken sufficiently often and at random; attaining reasonably accurate profiles at a lower overhead may become possible.

We came up with two possible strategies to decompose a program slice. Initial idea were to select random program points from the program slice during sampling. This technique was naive but it sampling would likely all program points in the slice. An alternative strategy would be to group statements in the slice based on a logical relation between them. An example of such logical relation is control flow relation, although other relations such as data-flow or a combination of these are also feasible candidates.

The former idea of decomposition may have some advantages. Due to random selection of the statements in a part, these statements are likely to be spread across the slice. These statements are also likely to be spread across different control flow paths. If a part of the slice is selected and instrumented, the probability that one or more instrumented statements are in the current execution flow of the program is high. Thus simple profile questions like “Is this statement ever executed?” can be easily answered. However, a disadvantage is that the amount of information collected is likely to be low and generally only sufficient for asking profile or monitoring questions related to individual program points. The profile questions that require implicit path information are harder to answer without significant instrumentation overhead and sampling cost.

The latter approach for decomposing a slice is more likely to help in answering better monitoring/

profiles questions. The relationship between statements that constitute a part of the slice facilitates answering path-related questions that build on that logical relation. For example, the logical relation "control flow" would facilitate answering questions such as "What are the frequently executed paths in this program?", "What are the major bottlenecks on a given path?", etc.

The rest of the sections provide details on specific steps of the approach. Section 3.2.1 explains and formalizes the notion of slice fragments. In Section 3.2.2, we discuss properties of slice fragment population and the significance of its *cover* (Section 3.1). The above two sections are illustrated with examples. The sampling and instrumentation techniques are discussed in the Section 3.2.3.

3.2.1 Computation of slice fragments

First, we compute program slice with relevance to a slicing criterion as depicted in Figure 2.2. Such program slice can be computed using any of the numerous techniques proposed in the literature (cf. [37]). The discussion of program slice computation is orthogonal to the scope of this thesis. After this, we prune the control-flow graph (Section 2.3), corresponding to the original program, to limit program points from program slice, while still maintaining the implicit control structures present in between them in the original graph. Multiple edges between the same nodes of the modified graph, if any, are reduced to a single edge.

This computed graph is termed as *slice-pruned control-flow graph*. Figure 3.1 shows the *slice-pruned control-flow graph* for the forward program slicing example in Figure 2.2. The computation of this graph is an important step geared towards the generation of slice fragments. We formally define **slice-pruned control flow graph** as follows:

Definition 3.2.1 *A* **slice-pruned control flow graph** *for a given control flow graph* $G_{cfg} = (V, E, v_0)$ *and forward static slice* $\mathcal{S}$*, is defined to be the graph* $G_s = (V', E', v_0, \mathcal{S})$ *where:*

- V' *is a set of nodes representing slice statements i.e.* $\forall v \in V', v \in \mathcal{S}$*,*
- $E' = \{(v_i, v_j) \mid (v_i, v_j) \in E$*, and* $v_i, v_j \in V'\}$
  $\cup \{(v_i \rightarrow v_j)$ *such that there exists a path* $< v_i, v_1, v_2, \ldots, v_k, v_j >$ *in* G_{cfg}*, where* $v_i, v_j \in V'$ *and* $v_1, v_2, \ldots, v_k \notin V'$*, and*

- v_0 *is the special entry node*

A crucial part of our approach is to group these program points such that each group *slice fragment* – is a logically related set with respect to a given property of interest. The construction of slice fragments is as follows. A depth-first search beginning at the root node (identified by slicing criteria) of *slice-pruned control flow graph* is kicked off. During this search we record each root node to leaf node sequence of the pruned graph as a slice fragment. As a consequence, the *back edges* encountered during loops, recursion and method return are ignored during the computation. This is to ensure that fragments are acyclic paths.

The six slice fragments computed for the graph in Figure 3.1 are shown in Figure 3.2. Consider the slice fragment 1 in Figure 3.2: it is a sequence $\langle entry, 5, 6, 8, 10\rangle$. Each node and its successor in this sequence is part of the edge set E' of the pruned graph. During the computation of fragments, the back edges, if any (due to loops, recursion or method return) are ignored. Some properties worth mentioning are that $entry$ node is included in all fragments and that the fragments do not contain duplicate vertices.

Based on the above, we have the following definition of *slice fragment*.

Definition 3.2.2 *A* **slice fragment** δ_{G_s} *of a slice-pruned control flow graph* $G_s = (V', E', v_0, \mathcal{S})$ *is a sequence* $\langle v_0, v_1, v_2, \ldots, v_n\rangle$ *where:*

- $v_0, \ldots, v_n \in V'$*, and* $v_i \neq v_j$*, where* $0 \leq i, j \leq n$ *and* $i \neq j$*,*
- *for any* i*, where* $1 \leq i \leq n-1$*, either* $(v_i, v_{i+1}) \in E'$*, or there exists a path in* G_s *from* v_i *to* v_{i+1} *such that all vertices on this path belong to the set* $\{v_1, \ldots, v_{i+1}\}$*, and,*
- *either* $\nexists v \in V'$ *such that* $(v_n, v) \in E'$*, or* $\forall v : (v_n, v) \in E' \implies v \in \{v_1, \ldots, v_{n-1}\}$

A *slice fragment* captures partial order(s) of implicit control relations between statements in the program slice. The construction of the fragment ensures the properties mentioned in definition. Note that this definition could be extended to relations other than control-flow, such as data-flow or combinations of both.

3.2.2 Normalizing slice fragment population

The computed slice fragments ensure that they together encompass all the program points in the slice of interest. No individual slice fragment can be larger than the slice itself. This is formally captured in the following lemma. The following observation follows directly from the definition of slice fragment.

Lemma 3.2.1 *Let δ_{G_s} be a slice fragment of a slice-pruned control flow graph $G_s = (V', E', v_0, \mathcal{S})$. Then, $|\delta_{G_s}| \leq |V'|$.*

Proof: According to Definition 3.2.2, (i) each node in δ_{G_s} belongs to V', and (ii) all the nodes in δ_{G_s} must be distinct. The lemma follows immediately.

3.2.2.1 Cover and Reduced Cover

The generated slice fragments form the population for sampling. The fragments in the population are more often than not likely to be subsumed by one another - program points of a fragment are completely encompassed within another. This is especially true in our case where typically the program points are common across many control structures. To identify this fragment populations which completely encompass program points in a slice, we define the concept of *cover*. In other words each statement in the program slice is *covered* by at least one slice fragment of a *cover*. Trivially, the initial population is a *cover*.

Definition 3.2.3 *Given a slice-pruned control flow graph $G_s = (V', E', v_0, \mathcal{S})$, we define a* **cover** *of G_s, denoted by Θ_{G_s}, to be a set of slice fragments of G_s, such that for each $v \in V'$, there exists $\delta \in \Theta_{G_s}$ such that $v \in \delta$.*

Including all the fragments in the population, may cause selection of fragments which subsume one another in a sample. This in turn may prove unproductive with regard to a chosen sampling strategy. This motivates the following definition.

Definition 3.2.4 *A cover Θ_{G_s} of a slice-pruned control flow graph G_s is called a* **reduced cover** *if there do not exist $\delta, \delta' \in \Theta_{G_s}$ such that each element of δ is also an element of δ'.*

For example, in Figure 3.2, fragments 1, 2, 3 and 5 together form a reduced cover of the slice depicted in Figure 3.1. Note however, that fragments 1, 2, 3, 5 and 6 together form a cover, but not a reduced cover of the slice.

For more on reduced cover and its construction refer Appendix A. One point to note here is that a cover does not associate any order among its fragments and therefore the reduced cover algorithm disregards the need to preserve or account for any order or priority of fragments in the resulting reduced cover.

3.2.3 Sampling fragments for statistical inference

To address the need of profiling and monitoring tasks that are the focus of this work, the cost of an individual monitoring step must remain fairly low, but a large number of such steps can be applied [8, 31] to gain profile information. The main idea is to pick a random sample of slice fragments periodically from the lot. The information obtained over a period of time from this sampling process can then be used to analyze the program with respect to the property under consideration.

One approach would be to perform *simple random sampling done without replacement*, that is, a member of the population is not chosen more than once in a sample. This approach is intuitively appropriate as we have minimum advance information about the population of slice fragments.

Another approach to manage a large slice fragment population, would be group them into few categories and then apply *stratified* sampling over these categories. This would give a better control over the captured profile information.

Following are some of the possible ways in which slice fragments can be "grouped".

- If the program slice is large and spans across many modules, then we could split up the fragments at module level and execute sampling based instrumentation for each of these modules. This granularity could be varied to work at the level methods or applications.

- Another option is to group fragments in the population by length, providing user better control over the amount of instrumentation.

If some statements are common to a large number of slice fragments, they are likely to be selected

more often than others, leading to repeated profile of such points. Also, if the statements in a sample of fragments are parts of loops or recursion, the monitoring overhead is likely to be huge. To alleviate this problem, we could have a code check to collect no more than x profiles over a period of y samples.

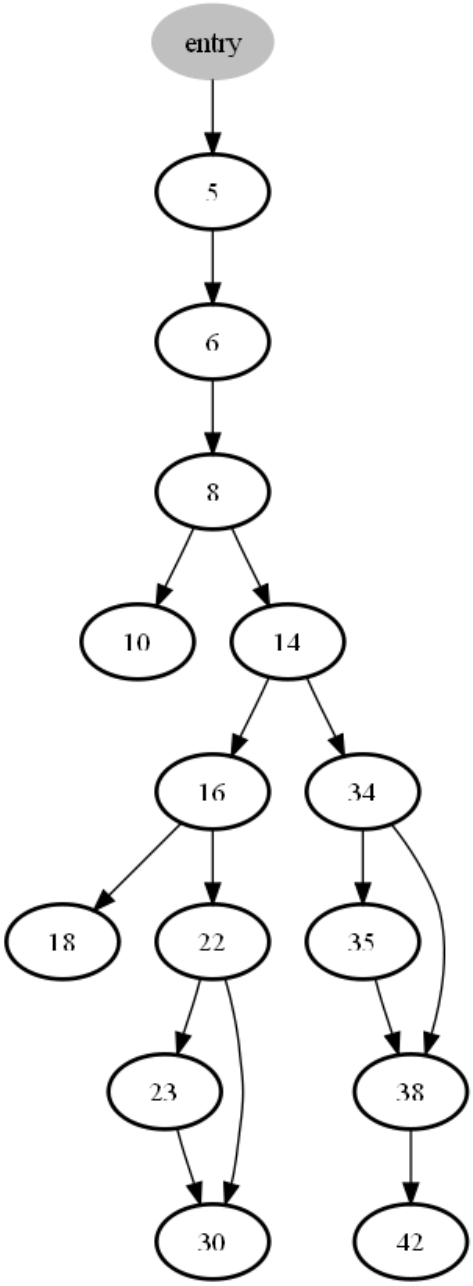

Figure 3.1 Slice-pruned CFG for program in Figure 2.2 with respect to slicing criterion <5, $userName$>. The numbers on the nodes correspond to the program line numbers.

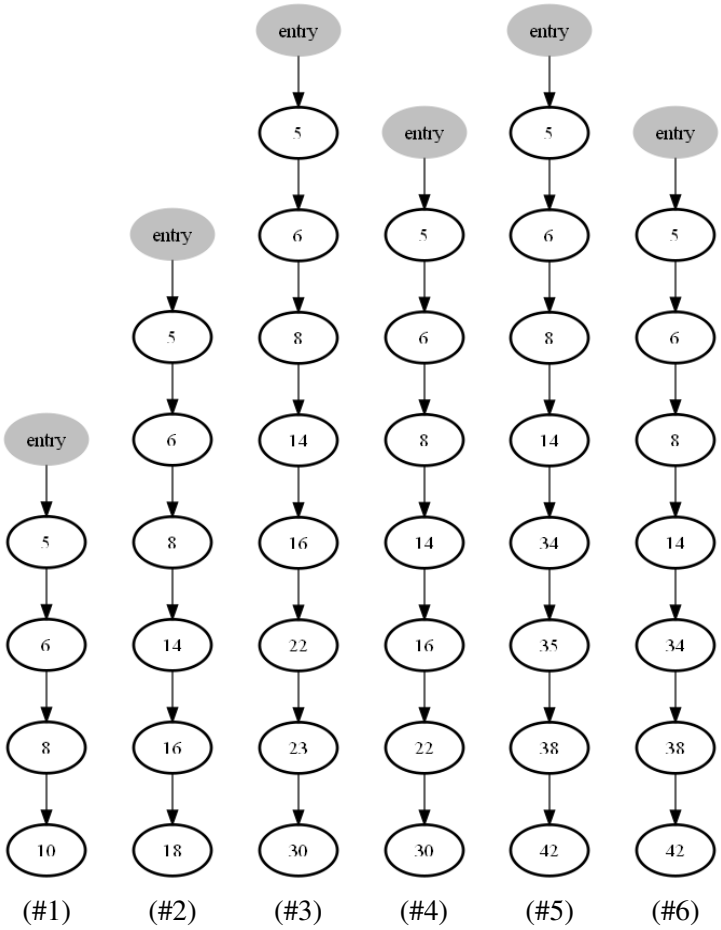

Figure 3.2 Slice fragments for the slice-pruned control-flow graph shown in Figure 3.1.

CHAPTER 4. EVALUATION

In this chapter, we discuss the details of the experiment which depict the properties of our approach. First we look into the experimental setup and tools used for the evaluation. The rest of the chapter sections present the potential utility of the approach. To that end, we analyze two properties. The first property of interest is whether, for a representative set of programs, our slice decomposition technique produces a *statistically significant* population of slice fragments. Here, *statically significant* implies that the population of constituent elements - the slice fragments, is large in number and average length of a fragment is a small subset of program points for the property of interest. This is important to keep the cost of instrumentation per sample to be low, but at the same time each of the samples should be able to capture relevant information. In our approach, as fragments tend to share a lot program points among them, a lower statistical significance may result in negligible gains at runtime. This property is a necessary pre-condition for applying any random sampling technique that we mentioned in Section 3.2.3. The empirical assessment of this property is described in Section 4.3. Second, we are also interested in exploring the reduction in scope that our approach helps achieve for typical programs. The empirical assessment of this property is also presented in Section 4.3. To study the properties of our technique, we simulated a random sampling process on the reduced population to determine the number of samples necessary to cover the population. This study is discussed in Section 4.4. In the last section of the chapter, Section 4.5, we detail the typical runtime and space overhead of our approach when compared to the full-slice instrumented version of the program.

4.1 Environment and Tools

To show the feasibility of our technique, we implemented our slice fragment computation and reduced cover construction algorithms as a stand-alone tool. Our tool is implemented using the slicing

functionalities provided by the IBM T. J. Watson Libraries for Analysis (WALA) [44]. WALA is a static analysis framework for Java bytecode, and provides a rich set of APIs for static analyses.

All experiments were conducted on a Dell Precision workstation with a 3.20GHz Intel Pentium D Processor and 2 GB of RAM using Sun JDK version 1.5_06 that was limited to use at most 1.5GB of heap space. WALA requires the input classes to be Java 5 bytecode compliant, therefore, all candidate projects were compiled under Sun JDK 1.5.0_06. In all the experiments, core Java libraries were excluded from the analysis.

Table 4.1 Static characteristics of subjects

Subjects	Number of Classes	Number of Methods	Bytecode size (in KB)
jaxen	217	1153	389
xstream	331	1519	774
nanoxml	24	541	35
jlex	27	133	88
spec/compress	12	33	18
spec/mtrt	25	470	32
spec/jess	192	1061	67

4.2 Subject Programs

For the experiments, a variety of subject programs were selected from different sources. We selected some open source programs, namely: *jaxen* - a XPath engine for Java [41], *xstream* - a library to serialize objects to XML [45]. *jlex* - a lexical generator for Java [42], Another program, namely *nanoxml* was sourced from the software-artifact infrastructure repository (SIR) [22] maintained by researchers at University of Nebraska, Lincoln. *Nanoxml* is a simple SAX parser. In addition, *mtrt*, *compress* and *jess* benchmarks from SPECjvm98 benchmarks [43] were also chosen. Table 4.1 shows some static properties of these programs.

Table 4.2 Results of slice decomposition and reduction algorithm (Algorithm 1) on subject programs

Subjects	Slicing criteria	Slice size (S)	Raw Slice Fragments			Reduced Slice Fragments			
			POP	Fragment length (average)	Fragment length (% of S)	POP	Reduction (% of Raw POP)	Fragment length (average)	Fragment length (% of S)
jaxen-1	XPath expression	232	79	12	5.17	50	41.42	13	5.60
jaxen-2	XPath expression	111	41	16	14.41	23	43.90	16	14.41
xstream	<Object> for XML conversion	173	9482	31	17.91	322	96.6	26	15.02
nanoxml	<XML> to be processed	140	1417	43	30.71	97	93.15	42	30
jlex	<File> for lexical analysis	67	28	16	23.88	14	50	18	26.86
spec/compress	*benchmark*	397	3267	94	23.67	46	98.59	79	19.899
spec/mtrt	*benchmark*	130	269	50	38.46	37	86.24	56	43.07
spec/jess	*benchmark*	42	53	15	35.71	13	71.69	15	35.71

4.3 Assessment of statistical significance

The prototype tool was used to generate slice fragments for subject programs mentioned in the previous section. The tool first computed the entire set of fragments and then applied the reduced cover algorithm (Algorithm 1) discussed in Appendix A. to create a reduced population of fragments. For this empirical study, we did not consider the algorithm discussed in Appendix A.1.1 for population reduction as our simpler algorithm was able to handle the number of generated fragments.

Table 4.2 shows the results of the slice decomposition process to generate raw population and its subsequent reduction. For each program, the slicing criteria selected is mentioned in the second column. The slicing criterion was selected to be representative of its typical usage.

The size of the generated slice is shown in the third column of the figure. All subject programs, when decomposed, showed a statistically significant population of slice fragments. In the Table 4.2, *jaxen-1* and *jaxen-2* refer to the same program but differ in the way slice was instantiated, that is, the slicing criteria was different in each case. It was observed that program slice of programs which were almost of the same size, showed huge disparity in corresponding raw populations (as seen in case of *xstream* vs. *nanoxml*, *nanoxml* vs *spec/mtrt*), primarily because of different control structures.

The reduced population computed was also found to be statistically significant, except in the case of *spec/jess* and *jlex*. *spec/compress* showed the largest drop in fragment population on application of the reduced cover algorithm. This benchmark implements a compression algorithm as a straight line code in two methods with very few method calls and large number of small branches. The large number of branches led to significant drop in population as most of them were subsumed. As there were no significant bifurcation in the program's control structure, number of resulting fragments remained low.

The least reduction was observed in *jaxen-1*. To begin with, *jaxen-1* had a relatively lower number of slice fragments. In addition, the implicit control structures within the program did not share much common code for them to be eliminated during computation of reduced cover.

With the exception of the benchmark programs *spec/mtrt* and *spec/jess*, the average fragment length was a small percentage of the slice (well within 30%). In *mtrt* benchmark, the control flow is within a long method `RenderScene` with a number of small branches, leading to longer slice fragments.

We also observed that slice size was determined mostly by the slicing criteria and slice configuration

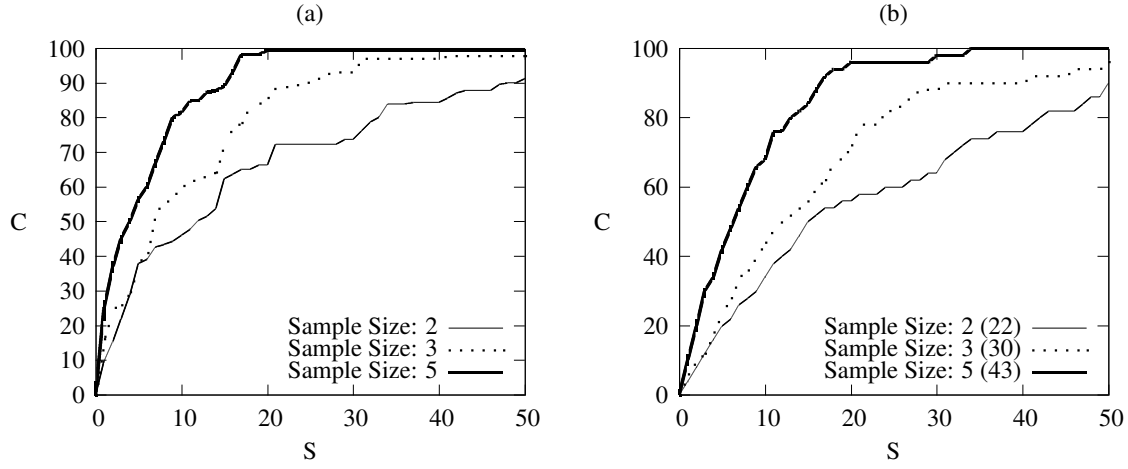

Figure 4.1 ***jaxen-1* statistical analysis (C = coverage, S = samples):** a) slice statement coverage b) slice fragment coverage

options and not by the size of the program, which was in line with our intuition.

4.4 Assessment of sampling and coverage

In this section, we empirically evaluate (i) the rate at which the slice fragments are *covered* during random sampling - *slice fragment coverage*, (ii) the rate at which slice statements are *covered* by the samples from reduced population - *slice statement coverage*, and (iii) the average number of unique statements per sample. To this end, we simulated simple random sampling on the reduced population. The results of these are presented in Figures 4.1-4.6 for most of the subject programs. The part *(a)* of these figures depicts *slice statement coverage*, while part *(b)* shows *slice fragment coverage* of the corresponding subject program.

Across all programs, it was observed that slice statements were *covered* in less number of sampling iterations when compared to slice fragments. Slice fragments tend to share a lot of common statements between them and therefore coverage of statements during sampling need not lead to coverage of fragments. Moreover, the random selection process is on population of slice fragments and not on slice statements. Due to this, it can also be observed that in most cases, the graph of statment coverage appears slightly skewed when compared to that of fragment coverage. The results obtained were in

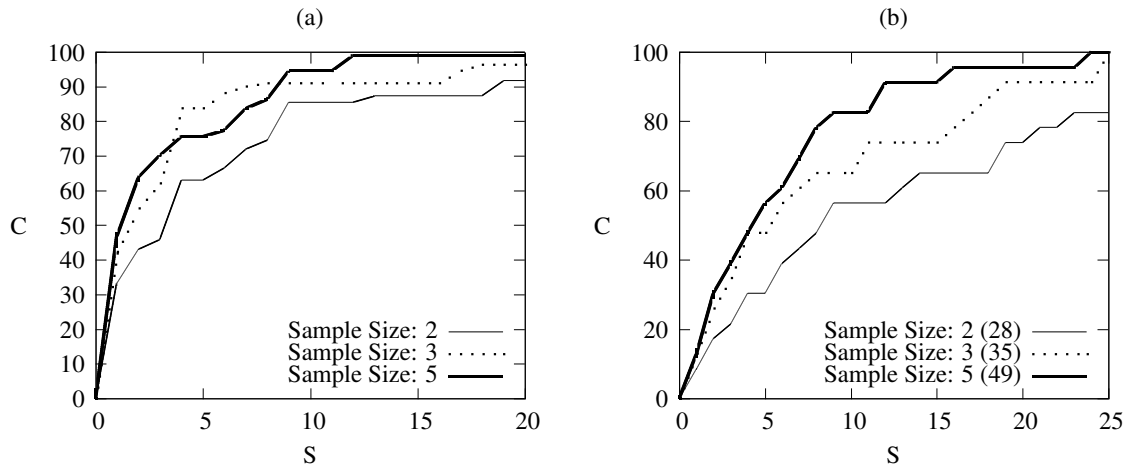

Figure 4.2 ***jaxen-2* statistical analysis (C = coverage, S = samples):** a) slice statement coverage b) slice fragment coverage

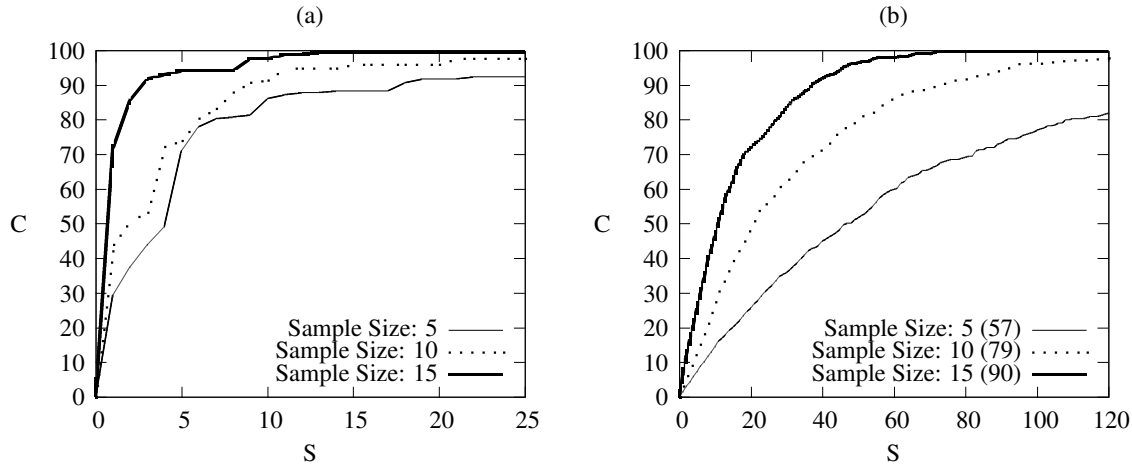

Figure 4.3 ***xstream* statistical analysis (C = coverage, S = samples):** a) slice statement coverage b) slice fragment coverage

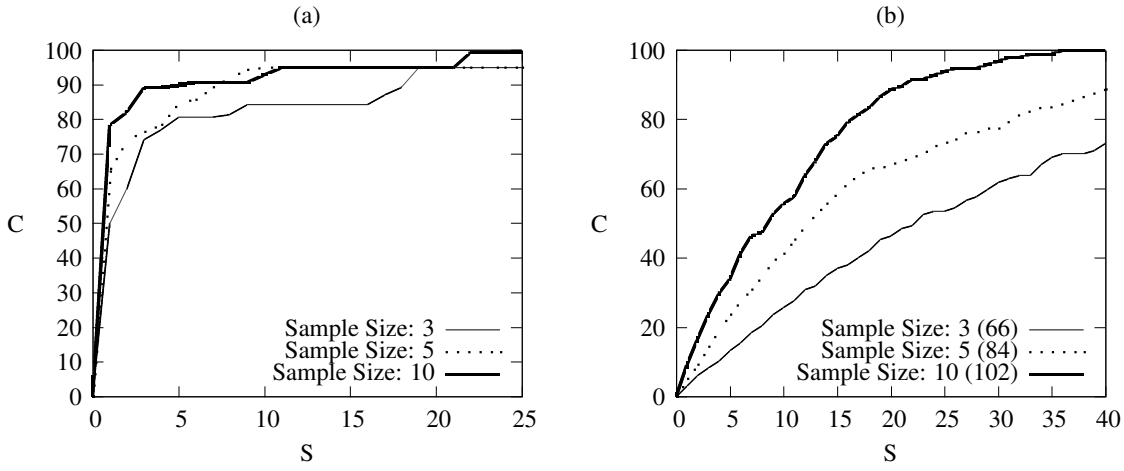

Figure 4.4 ***nanoxml* statistical analysis (C = coverage, S = samples):** a) slice statement coverage b) slice fragment coverage

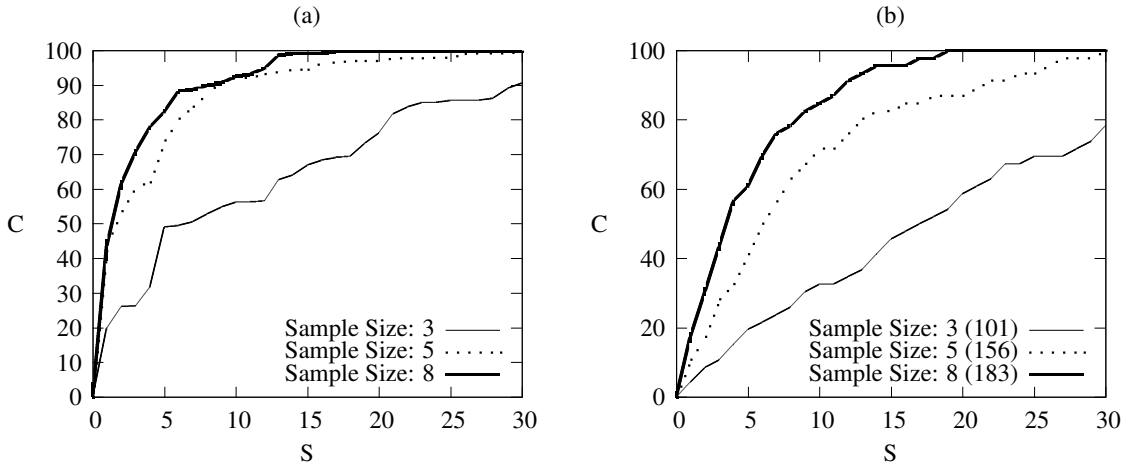

Figure 4.5 ***spec/compress* statistical analysis (C = coverage, S = samples):** a) slice statement coverage b) slice fragment coverage

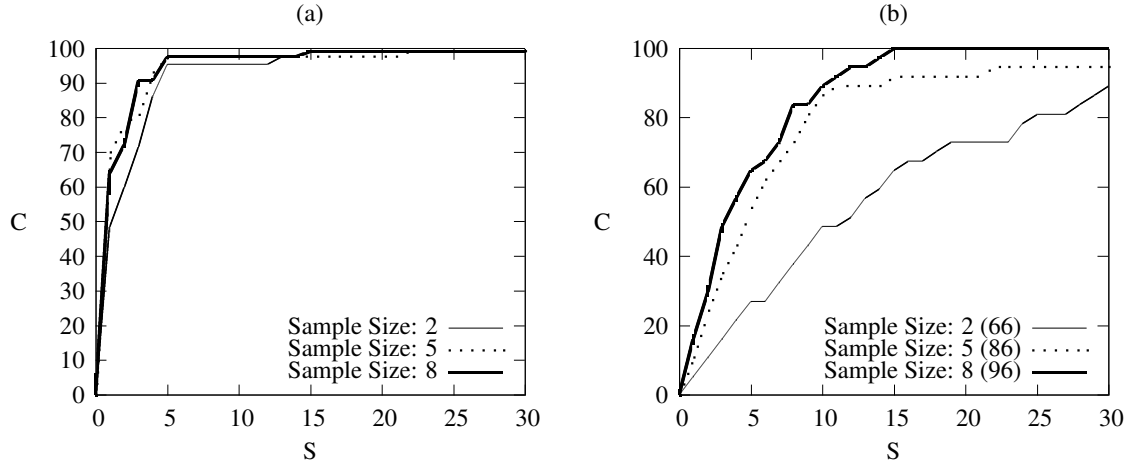

Figure 4.6 ***spec/mtrt* statistical analysis (C = coverage, S = samples):** a) slice statement coverage b) slice fragment coverage

line with this intuition. The disparity in coverage of statements and fragments over sample iterations was most pronounced in case of *xstream*. It showed that under different sample sizes, sampling 25 times was sufficient to reach near optimal statement coverage, whereas 120 samples were required to observe the same in case of slice fragments. *jaxen-2* showed skewed fragment coverage, due to small population. In part *(a)* of the Figures 4.1-4.6, the numbers beside the sample sizes in the labels (with round brackets) indicate the corresponding average number of unique statements per sample. When the sample size chosen was relatively lower, these unique statements were observed to be significantly small subsets of their corresponding program slices.

4.5 Runtime behavior

In this section, we record the observed runtime behavior for a few of the subject programs. The purpose of this experiment was to study the overhead of our approach, by recording the runtime and space overhead values for each of the programs over sample iterations when compared to the full-slice instrumented version of the program.

During every iteration, we first randomly select a sample of slice fragments from pre-computed reduced population of slice fragments. The statements from these fragments are then collected, nor-

All slice statements instrumented (FULL)	Runtime statement execution			Increase in size (KB)
	Input 1	Input 2	Input 3	
	104	170	117	260

(A)

Iteration	Sample size = 2			Sample size = 3			Sample size = 5		
	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3
5	11	51	1	22	26	14	13	16	3
10	5	5	57	5	5	1	11	15	3
15	7	7	15	15	17	12	22	29	24
20	1	1	4	18	58	6	15	19	9
25	7	10	1	11	13	13	15	19	7
30	1	1	46	5	15	15	1	1	5
35	13	13	1	8	8	17	15	27	3
40	7	7	1	12	12	51	11	13	24
45	11	24	1	5	5	7	17	19	13
50	17	22	17	1	1	1	3	3	3
Avg.	7.12	10.26	8.98	9.82	13.7	14.22	15.7	24.18	18.82
% FULL	93.15	93.96	92.32	90.55	91.94	87.84	87.90	85.77	83.91

(B)

Iteration	Increase in size (KB)		
	2	3	5
5	24	72	52
10	40	24	72
15	24	48	88
20	24	44	56
25	24	68	52
30	24	32	44
35	20	44	52
40	24	44	88
45	20	28	108
50	76	28	56
Avg.	33.36	49.68	66.32
% FULL	87.1	80.89	74.49

(C)

Figure 4.7 ***jaxen-1* runtime analysis:** Runtime statement execution (B) and space overhead (C) under various sample sizes for selected inputs. (A) has these details for full-slice instrumented version of the program.

malized and then a simple instrumentation was performed - at the beginning of each statement, a log statement was added to record the execution of the original statement. After this instrumentation, the updated application classes are put on priority on classpath, so as to give them precedence during class loading/ execution. The program is then run with the new classpath set, on different inputs (valid/ exception cases). The actual log printed for each of the inputs is recorded. It has to be noted that the number of statement logs may be well over the number of statements selected for instrumentation for a given iteration. This is because statements may be executed more that once, for example, due to loop constructs and multiple invocations of a method which has some of its statements instrumented. The above operation is repeated for various inputs under various sample sizes of slice fragments. In addition, at the end of each iteration, the space overhead added by the instrumentation is also noted. The space overhead is computed for during each iteration of sampling and is not dependent on the actual input to the program.

All slice statements instrumented (FULL)	Runtime statement execution			Increase in size (KB)
	Input 1	Input 2	Input 3	
	26	26	99	140

(A)

Iteration	Sample size = 2			Sample size = 3			Sample size = 5		
	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3
5	14	14	14	5	5	13	21	21	35
10	5	5	26	5	5	55	5	5	26
15	1	1	15	5	5	26	16	16	64
20	1	1	7	8	8	16	14	14	67
25	4	4	12	5	5	14	19	19	72
30	4	4	6	1	1	13	5	5	56
35	14	14	22	5	5	58	1	1	4
40	5	5	14	21	21	26	8	8	24
45	1	1	48	4	4	24	7	7	19
50	4	4	7	5	5	15	4	4	58
Avg.	4.56	4.56	13.6	8.68	8.68	24.	10.42	10.42	33.94
% FULL	82.4	82.4	86.26	66.61	66.61	75.75	59.92	59.92	65.71

(B)

Iteration	Increase in size (KB)		
	2	3	5
5	60	52	88
10	44	44	48
15	16	48	96
20	16	48	68
25	48	44	96
30	48	28	48
35	60	40	28
40	44	72	48
45	28	56	80
50	48	44	56
Avg.	36	50	64.72
% FULL	74.28	64.28	53.77

(C)

Figure 4.8 ***jaxen-2* runtime analysis:** Runtime statement execution (B) and space overhead (C) under various sample sizes for selected inputs. (A) has these details for full-slice instrumented version of the program.

	Runtime statement execution		Increase in size (KB)
	Input 1	Input 2	
All slice statements instrumented (FULL)	238	262	532

(A)

Iteration	Sample size = 5		Sample size = 10		Sample size = 15	
	Input 1	Input 2	Input 1	Input 2	Input 1	Input 2
5	92	110	98	110	104	119
10	77	86	101	113	144	162
15	65	77	138	156	129	141
20	117	126	107	125	110	128
25	86	98	101	113	144	162
30	122	146	156	174	107	125
35	89	104	86	104	138	156
40	101	113	122	140	122	146
45	86	98	113	137	116	134
50	92	104	116	134	113	131
55	80	92	129	141	113	131
60	74	83	159	183	137	161
65	129	141	98	110	141	159
70	110	128	144	162	174	198
75	80	86	119	137	119	137
80	68	80	147	165	116	134
85	126	138	110	128	156	174
90	122	146	113	131	128	146
95	104	122	113	131	122	146
100	128	152	122	146	150	168
105	98	116	131	152	138	156
110	89	104	110	128	144	162
115	95	107	98	116	171	195
120	92	104	86	104	171	195
Avg.	103.21	118.71	120.86	138.69	135.7	155.12
% FULL	56.63	54.69	49.21	47.06	42.98	40.79

(B)

Iteration	Increase in size (KB)		
	5	10	15
5	192	236	292
10	204	252	264
15	164	256	300
20	184	252	292
25	172	248	304
30	188	268	264
35	224	192	296
40	180	212	288
45	200	236	300
50	200	264	268
55	216	240	280
60	188	252	296
65	192	212	312
70	196	268	312
75	184	256	288
80	180	224	284
85	216	232	264
90	164	252	300
95	192	240	276
100	164	260	288
105	156	244	252
110	176	244	300
115	180	280	300
120	184	168	292
Avg.	188.46	241.23	281.24
% FULL	64.57	54.65	47.13

(C)

Figure 4.9 ***xstream* runtime analysis:** Runtime statement execution (B) and space overhead (C) under various sample sizes for selected inputs. (A) has these details for full-slice instrumented version of the program.

All slice statements instrumented (FS)	Runtime statement execution			Increase in size (KB)
	Input 1	Input 2	Input 3	
	10878	51803	2349	112

(A)

Iteration	Sample size = 3			Sample size = 5			Sample size = 10		
	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3
5	2343	1359	584	9974	4848	2066	10305	5067	2168
10	2445	1490	643	9703	4701	2000	5019	3266	1405
15	8928	4079	1740	9693	4686	2003	8851	4103	1751
20	8905	4082	1743	8993	4095	1756	10866	5488	2345
25	8905	4058	1731	2572	1525	664	8912	4114	1762
30	2543	1531	670	8527	3838	1637	4534	2906	1254
35	8286	3653	1556	3820	2421	1039	9694	4687	2004
40	9435	4515	1918	3920	2523	1088	9660	4662	1993
Avg.	4514.6	2235.5	957.7	5794.6	2874.2	1230.8	7153.6	3533.3	1512.1
% FS	58.49	95.68	59.22	46.73	94.45	47.60	34.23	93.17	35.62

(B)

Iteration	Increase in size (KB)		
	3	5	10
5	76	88	96
10	76	80	96
15	68	84	92
20	84	96	96
25	80	84	96
30	84	84	100
35	68	80	88
40	60	84	92
Avg.	75.6	84.5	92.9
% FS	32.5	24.55	17.05

(C)

Figure 4.10 ***nanoxml* runtime analysis:** Runtime statement execution (B) and space overhead (C) under various sample sizes for selected inputs. (A) has these details for full-slice instrumented version of the program.

Figures 4.7- 4.10, provide details of execution and space overhead for the select programs. Each of these figures have three parts. Part A) provides the execution data obtained when the all of the slice statements were instrumented. Part B) provides execution overhead is measured in terms of number of instrumented logs recorded. (it is assumed that the overhead per instrument statement is constant) Part C) provides space overhead recorded during sampling process. The measurement of execution data in terms of statements as we will see, has the potential to reveal more information about behavior of program for a given input.

In all cases, it was observed that as the sample size was increased, the corresponding logging of instrumented slice statements also increased. If more fragments are selected during sampling, more statements are likely to be instrumented. It was also observed that in certain iterations, for certain inputs, there was hardly any logging. In case of *jaxen-1* and *jaxen-2*, as seen in Part B) of Figure 4.7 and Figure 4.8 there are a lot of entries which have **1** in them. This is most probably due to logging of just the instrumented root slice statement). This also implies, there was either insignificant overhead in the above case.

In addition, in these above two cases, the statement execution overhead when compared to the full-slice instrumented versions of the respective programs was significant (well above 60%). The runtime gains obtained in case of *xstream* (Figure 4.9) were not too significant, but it was observed that it did not change too much based on the sample sizes. On the other hand, *nanoxml* (Figure 4.10) showed noticeable variation in execution overhead obtained based on the sample sizes.

The space overhead was observed to increase with increasing sample sizes, which was in line with our intuition. Though space overhead is not important aspect of out approach, it provides an indication of amount of instrumentation happening during sampling iterations.

It was also observed that at runtime the slice fragments were part of the execution flow. It is to be noted here that he computation of slice fragments which is done statically is conservative. As a result, the some fragments of the population may never actually be detected during program runtime. In Figure 4.11 we see that captured fragments, increased with increasing sample size, except in case of *jaxen-2*. As its slice fragment population was small, the increase in sample size did not reflect any changes between sample sizes.

Iteration	Sample size = 2			Sample size = 5			Sample size = 8		
	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3
40	11	8	9	20	14	10	16	18	16

jaxen-1

Iteration	Sample size = 2			Sample size = 3			Sample size = 5		
	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3	Input 1	Input 2	Input 3
40	30	25	25	30	25	25	30	25	25

jaxen-2

Iteration	Sample size = 5		Sample size = 10		Sample size = 15	
	Input 1	Input 2	Input 1	Input 2	Input 1	Input 2
60	9	9	26	26	34	34

xstream

Figure 4.11 ***Slice fragments captured*:** The tables show the slice fragments captured at runtime over sampling iterations for some inputs

CHAPTER 5. USE CASES

In this chapter, we present a couple of use cases in which our approach would be appropriate to be applied.

5.1 Selective Call Path Monitoring

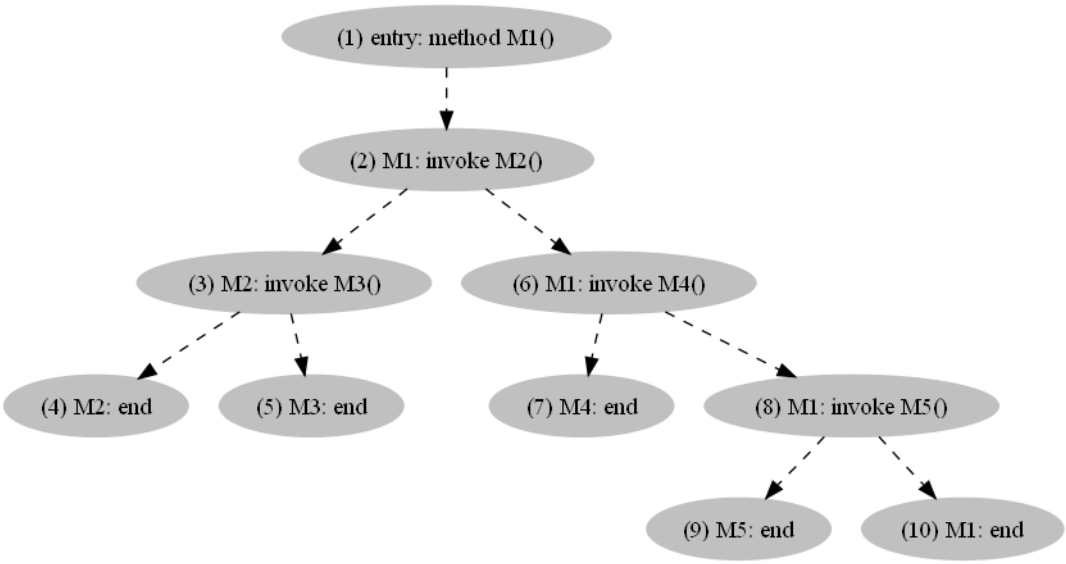

Figure 5.1 Slice-pruned call graph of a program

Previous work have found the call path profiles, among other things, to be useful in performance analysis [30, 17]. A call path profile captures nested sequence of call encountered at execution time. This in turn can be used to capture those nested sequence of calls which consume most program execution. It can help us identify opportunities for future program enhancements.

Using our approach, it is possible to get a more selective version of such a call path profile. A call path profile in the context of a program slice helps the analysis focus better on the context(slicing

criteria) at hand and eliminate irrelevant call nodes during analysis.

We illustrate the application with an example. Figure 5.1 shows a example call graph of a program. Consider it to have been pruned to contain statement from the slice. Also, the arrow-dotted lines indicate non-method invocation statements to have been scoped out for this application.

When we generate slice fragment population on the graph in the figure, we have the following fragaments:

- (1), (2), (3), (4)
- (1), (2), (3), (5)
- (1), (2), (6), (7)
- (1), (2), (6), (8), (9)
- (1), (2), (6), (8), (10)

These fragments represent the partial order of call paths that can occur in the program. Applying our approach further, one should be able to get performance analysis on partial order of call sequences. This in turn can help us zero in on set of call sequences having possible performance issues.

5.2 Distributed Program Monitoring

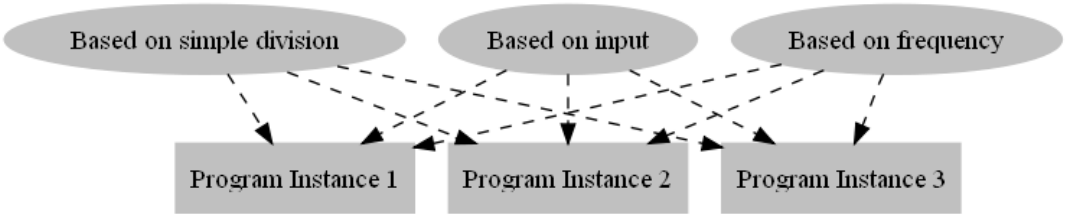

Figure 5.2 Distribution of monitoring tasks across different instances of a program may be possible based on different dimensions - *simple grouping of fragments*, *program input*, *frequency of execution*

Another potential application of our approach could be in the context of GAMMA system based on Software Tomography. An adaptation of this approach that utilizes our technique could be to distribute

slice fragments of the population into N groups. Ideally, there should be equal distribution of the fragments among these N groups. Like in the GAMMA system, N instances of software can then be created, where each instance is assigned to a particular group of slice fragments. Such instances could then be instrumented at program points corresponding to the slice fragments in the assigned group.

There may be many different ways in which the above N groups can be constructed. Following are possible options which may be used, based on, which is also illustrated in Figure 5.2.

- simple distribution of slice fragments
- predicted frequency of slice fragments which will be executed
- length of the slice fragments

CHAPTER 6. RELATED WORK

This work primarily proposes using program slicing along with statistical sampling to monitor and profile programs. Specifically, we propose selecting subset of program slice entities for monitoring and profiling software, using random sampling based schemes. This chapter discusses the work most relevant to our own.

We share a similar goal with *software tomography* [8] as realized in the GAMMA system [31]; that is, a reduction of monitoring overhead in deployed software instances. The main idea in *software tomography* is to divide and allot the monitoring tasks across several instances of the software. It then collects the data from these instances to compute monitoring information for the complete application. MOP [9] is a runtime verification framework, which generates monitors from the specified properties and integrates it with the application. It uses decentralized indexing to reduce the overhead of monitoring at execution time. In contrast, we apply program slicing and statistical sampling to reduce the scope of similar tasks.

Arnold and Ryder [4] use a profiling framework combined with code duplication to reduce the instrumentation overhead. This framework samples the instrumented version of the code for bounded amounts of time to collect the required profiles from the program. On the other hand, our approach samples on the decomposed slice fragments, restricted to a property of interest, to compute the software profile.

We share the same objective as that of Santelices et al. [35] and Apiwattanapong et al. [3] which proposes to reduce the set of program entities to a smaller subset to decrease the monitoring overhead. Our approach differs in that it uses random sampling to obtain similar results.

Thin Slicing [36] proposes a selective notion of relevance based on a seed computation to reduce the scope of debugging and program understanding tasks. Though we share a similar goal, our approach is

oriented towards profiling and monitoring software.

Liblit et al. [26] propose a sampling infrastructure based on a Bernoulli process to gather information about a software from user executions with low overhead. Their main focus is on bug isolation using statistical analysis. Our primary focus is on low overhead monitoring of software using statistical sampling of fragments.

Leveraging recent advances in run-time systems, Dwyer et al. [13] propose adaptive online program analysis (AOPA) to reduce overhead of dynamic analyses. AOPA adaptively varies instrumentation to observe program behavior, assuming a reduced scope for analyses. In contrast, we use sampling on precomputed fragments for profiling.

Dynamic program slicing [1] proposes the concept of computing statements which actually affect the value of given variable for a specific input. These slices likely to vary depending on the program input. Our approach attempts to capture the profile corresponding to the set of fragments in a sample for various program inputs.

Younes and Simmons [40] propose a probabilistic method for verifying properties of discrete event systems. The key idea behind their work is that probabilistic models are used to reduce the verification efforts. They use probabilistic models to select a subset of the state space. While we share a similar insight, this work proposes sampling of decomposed slice fragments to reduce the number of instrumentation points to gather monitoring and profile information.

Luk et al. [28] propose *Pin*, as software system that performs runtime instrumentation of Linux applications. *Pin* performs instrumentation using a JIT (just-in-time) compiler. The input to this compiler is a native executable and not Java bytecode. *Pin* instruments the system-level programs irrespective of the underlying architecture of the system. Our work is in the direction of segregating the relevant portion of the program with the goal of avoiding instrumenting irrelevant portions of the program.

Hiniker et al. [21] study the problem of code selection for dynamic optimization of systems. They mainly address the problems of trace selection and excessive code duplication. They implement region-selection algorithms which rely on the *Pin* system to report the sequence of basic blocks executed by the program. The *trace-combination* algorithm incrementally builds the CFG for the program as it executes. Our approach uses slice decomposition to reduce the overhead of instrumentation.

CHAPTER 7. CONCLUSION

The main goal of this paper was to present program slicing as an instrument for monitoring and profiling tasks. In this regard, I have made following key technical contributions:

- the notion of slice fragments,
- computation of slice fragments,
- a use case of slice fragments for a statistical sampling-based instrumentation technique.

Our technique first uses slicing to narrow down the scope of the instrumentation to that of interest with respect to a property (expressed as slicing criterion). We then provide a method to further decompose the slice into (smaller) slice fragments. A subset of slice fragments are then instrumented for monitoring or profiling tasks. We also presented empirical results to validate that our technique can collect profiles at high assurance levels, at a significantly lower overhead.

There are certain obvious limitation of the current study. Not all monitoring problems can be reduced to monitoring of a program slice. It does not look into the details of instrumentation mechanism to collect the profiles from slice fragments.

Several interesting avenues remain to be explored, especially on various instrumentation techniques to collect the profile data. If correlation between slice fragments of the population and profile information gathered can be established, then it can be used to guide future instrumentation sampling.

An empirical study could be conducted for larger programs (perhaps with millions of line of code) to revalidate our current results on a representative subset of Java programs. An automated technique for determining an optimal sample size for programs would also complement our approach.

With the growing size, complexity, and adaptability of software systems both the instrumentation overhead as well as the need for monitoring and profiling is likely to increase. Our approach thus

provides a timely advance towards enhancing the scalability of monitoring and profiling processes to cope with these challenges.

APPENDIX A. Reduced cover

The appendix here discusses some algorithms for computation of reduced cover of slice fragments. The following work is the contribution of Mukul Bansal(bansal@cs.iastate.edu) and Dr. Hridesh Rajan(hridesh@cs.iastate.edu) and has been included here for the sake of completeness.

A.1 Reduced Cover Construction

A brute-force way to construct a reduced cover is to first compute all possible slice fragments for the slice, and then to delete, one at a time, those slice fragments in which all the vertices also appear in some other slice fragment. We now show how a much more elegant and efficient algorithm can be used to achieve the same result. One of the other desirable properties of this method is that it produces a reduced cover which is not much larger than the smallest possible (because it is based on a well-known approximation algorithm for the set cover problem [16]).

Let Θ denote the set of slice fragments generated. We now apply Algorithm 1 on Θ. The main idea here is to repeatedly identify a slice fragment from Θ that contains the largest number of uncovered vertices in the slice and add it to the set Θ', until all the vertices of the slice have been covered.

Algorithm 1 is in fact a well known heuristic (and approximation algorithm) for the set cover problem [23]. Here, the vertices of the slice form the elements of the universe, and each slice fragment in Θ can be viewed as a subset of this universe. It is possible to implement Algorithm 1 such that its time complexity is $O(\sum_{\delta \in \Theta} |\delta|)$, i.e. it is linear in the size of all the slice fragments in Θ (see [11]).

Let Θ^* denote the set of slice fragments returned by Algorithm 1. We claim that Θ^* must be a reduced cover for the slice.

Lemma A.1.1 *Θ^* is a cover of the slice.*

Algorithm 1 Generating a reduced cover for the slice

Require: The set of slice fragments Θ
1: Let $\Theta' \Leftarrow \emptyset$
2: Let V be the set of all vertices in the flow graph corresponding to the slice.
3: **for** each $v \in V$ **do**
4: Set $label(v) = false$
5: **end for**
6: **repeat**
7: $\alpha \in \arg\max_{s \in \Theta} |\{v \in s \colon label(v) = false\}|$
8: Add α to Θ'
9: **for** each vertex v in α **do**
10: Set $label(v) = true$
11: **end for**
12: **until** $label(v) = true$ for each $v \in V$
13: **Return** Θ'

Proof: Consider the set Θ. Since Θ consists of all possible slice fragments of the slice, Θ must be a cover of the slice. Algorithm 1 does not terminate until all vertices of the slice have been covered. In the worst case, this might entail adding all the slice fragments in Θ to Θ^*. Therefore, Θ^* will always be a cover of the slice.

Proposition A.1.1 *Θ^* is a reduced cover of the slice.*

Proof: By Lemma A.1.1 we already know that Θ^* is a cover of the slice. Therefore, for the sake of contradiction, let us assume that the cover Θ^* is not reduced. Then, there must exist $\delta, \delta' \in \Theta^*$ such that $v \in \delta \Rightarrow v \in \delta'$. There are two possible cases: (i) Algorithm 1 adds δ to Θ^* before it adds δ', or (ii) Algorithm 1 adds δ' to Θ^* before it adds δ. We analyze each of these cases separately.

Case (i): Let $\widehat{\Theta}$ denote the set Θ' in Algorithm 1 immediately before the addition of δ. Since Algorithm 1 adds δ before adding δ', all the vertices in $\delta' \setminus \delta$ must already be covered by $\widehat{\Theta}$. This implies that as soon as δ is added to $\widehat{\Theta}$, all the elements of δ' are also covered. Hence, Algorithm 1 would not add δ' to Θ^*. This case is therefore infeasible.

Case (ii): After the addition of δ' to Θ^*, all the elements of δ have already been covered. Therefore, Algorithm 1 would not add δ to Θ^*. This case is therefore infeasible.

Since neither of these two cases is possible, we have arrived at a contradiction. Hence, Θ^* must be a reduced cover.

We illustrate this algorithm using an example. Consider the slice fragments depicted in Figure 3.2.

Algorithm 1 takes these as input, and produces a reduced cover for the slice. At each step the algorithm chooses a slice fragment that covers the largest number of uncovered nodes. Thus, the algorithm first chooses the slice fragment 5 which is $\langle entry, 5, 6, 8, 14, 34, 35, 38, 42\rangle$. In the next step, fragment 3 encompassing $\langle entry, 5, 6, 8, 14, 16, 22, 23, 30\rangle$ is chosen. Similarly, fragment 2 gets picked up next. In the end, node 10 is the one not covered so far, resulting in fragment 1 being picked. It is easy to see that they form a reduced cover of the program slice.

A.1.1 Alternate Reduced Cover Construction

Observe that the algorithm seen above requires us to first compute the set of all slice fragments for the slice. In cases where the number of slice fragments is prohibitively large, we can use an add-on algorithm to reduce the number of slice fragments that need to be generated. Such an algorithm would begin with the slice-pruned control flow graph, and modify it by deleting edges. This produces a smaller graph, which will have fewer slice fragments.

The above algorithm may be more efficient but is effective in problems where monitoring "class of flows" in a program. The reduced population of slice fragments would be helpful in answering questions like:

Consider the following problem: Given a directed graph, find a smallest subset of edges in the graph that maintains all reachability relations between the vertices. This problem is known as the *minimum equivalent graph (MEG)* [29] problem[1]. As shown in the following proposition, solving the MEG problem provides a way to reduce the size of the slice-pruned control flow graph while still preserving the required coverage and connectivity properties.

Proposition A.1.2 *Given a slice-pruned control flow graph $G_s = (V', E', v_0, \mathcal{S})$, let $G' = (V', E'', v_0, \mathcal{S})$ be a minimum equivalent graph of G_s. Then, the set of all slice fragments of G' forms a cover of the slice G_s.*

Proof: Let Θ and Θ' denote the set of all slice fragments of G and G' respectively. We know that Θ is a cover of the slice. We will show that for any slice fragment $\delta \in \Theta$, there exists some slice fragment

[1] Also known as the *minimum equivalent digraph* problem.

$\delta' \in \Theta'$ such that $v \in \delta \Rightarrow v \in \delta'$. Given any $\delta \in \Theta$, let u, v be any two consecutive vertices in δ. Since G contains a path from u to v, by definition, G' must also contain a path from u to v. This is true for every consecutive pair of nodes u, v in δ; which implies that there must be a path, not necessarily simple, in Θ' with the same start and end vertices as δ, and which passes through all the nodes of δ. If we let δ' be the slice fragment corresponding to such a path, then $v \in \delta \Rightarrow v \in \delta'$.

The MEG problem is known to be NP-hard [15], however, several constant factor approximation algorithms exist for it (cf. [25]). These algorithms are guaranteed to produce, within polynomial time, a solution that is within some fixed percentage of an optimum solution. Note that the property stated in Proposition A.1.2 is monotone. Proposition A.1.2 therefore implies that any approximation algorithm for the MEG problem can be used to reduce the size of the flow graph, without adversely affecting our construction of a reduced cover for the slice.

BIBLIOGRAPHY

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*, pages 246–256, 1990.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

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

[4] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01*, pages 168–179.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00*, pages 1–12.

[6] V. Barnett. Sampling techniques: Principles and methods, 3rd edition. *Wiley, New York*, 1977.

[7] V. Barnett. Sample surveys: Principles and methods. *Oxford University Press*, 1991.

[8] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *PASTE '02,pp. 2–9*.

[9] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *OOPSLA '07*, pages 569–588.

[10] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00*, pages 439–448.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[12] S. Debray and W. Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, New York, NY, USA, 2002. ACM.

[13] M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *ICSE '07*, pages 220–229.

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

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[16] R. S. Garfinkel and G. L. Nemhauser. *Integer Programming*. John Wiley & Sons, New York, 1972.

[17] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[18] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *PLDI '07,pp. 256–265*.

[19] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher Order Symbol. Comput.*, 13(4):315–353, 2000.

[20] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI*, pages 156–164, 2004.

[21] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *MICRO '05)*, 2005.

[22] S. E. Hyunsook Do and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435, 2005.

[23] D. S. Johnson. Approximation algorithms for combinatorial problems. In *STOC '73: Proceedings of the fifth annual symposium on Theory of computing*, pages 38–49, 1973.

[24] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[25] S. Khuller, B. Raghavachari, and N. Young. Approximating the minimum equivalent digraph. *SIAM J. Comput.*, 24(4):859–872, 1995.

[26] B. Liblit, A. Aiken, and A. Zheng. Distributed program sampling. In *PLDI '03*, pages 141–154.

[27] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03*, pages 141–154.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200.

[29] D. M. Moyles and G. L. Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *J. ACM*, 16(3):455–460, 1969.

[30] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. *SIGPLAN Not.*, 44(10):175–190, 2009.

[31] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *ISSTA '02*, pages 65–69, 2002.

[32] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Symposium on Practical software development environments*, pages 177–184, 1984.

[33] T. Reps and T. Bricker. Illustrating interference in interfering versions of programs. In *2nd International Workshop on Software configuration management*, pages 46–55, 1989.

[34] T. W. Reps and W. Yang. The semantics of program slicing and program integration. In *TAPSOFT '89*, pages 360–374.

[35] R. Santelices, S. Sinha, and M. J. Harrold. Subsumption of program entities for efficient coverage and monitoring. In *SOQUA '06*, pages 2–5.

[36] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI '07*, pages 112–122.

[37] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[38] M. Weiser. Program slicing. In *ICSE '81*, pages 439–449.

[39] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[40] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV '02*, pages 223–235.

[41] Jaxen: Universal Java XPath engine. `http://jaxen.org/`. last accessed on 06/2009, *used ver 1.1.1 source code (04,2008).*

[42] JLex: A lexical analyzer generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`. last accessed on 06/2009, *used ver 1.2.6 source code (02/2003).*

[43] Spec jvm 98 benchmarks. `http://www.spec.org/jvm98/`. last accessed on 06/2009, *used JVM client suite benchmarks (11/1998).*

[44] T.J. Watson libraries for analysis. `http://wala.sourceforge.net`. last accessed on 07/2009, *used source code of ver 1.1 (01/2009) and 1.2 (07/2009).*

[45] XStream: Serialization library from objects to XML. `http://xstream.codehaus.org/`. last accessed on 06/2009, *used ver 1.3 source code (02/2008).*