Property-Aware Program Sampling

Harish Narayanappa[α] Mukul S. Bansal[β] Hridesh Rajan[α]

[α]Iowa State University [β]Tel Aviv University

[α]{harish,hridesh@cs.iastate.edu} [β]bansal@tau.ac.il

Abstract

Monitoring or profiling programs provides us with an understanding for its further improvement and analysis. Typically, for monitoring or profiling, the program is instrumented to execute additional code that collects 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.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Optimization; D.2.7 [*Software Engineering*]: Enhancement

General Terms Algorithms, Experimentation, Performance

Keywords static analysis, program slicing, slice fragments, profiling, sampling, instrumentation, property-aware monitoring

1. Introduction

An insight into the run-time behavior of a deployed software application provides potential opportunities for its improvement. Profiling or monitoring software provides such insights. The dynamic information gathered can be used in performance tuning of applications [8], coverage-based testing [20], analyzing the application's usability etc. For monitoring or profiling and for tasks such as bug detection [23, 15], continuous testing [1], dynamic optimization [4], it is often necessary to instrument programs.

Full instrumentation to collect data about all interesting program points reportedly causes between 10% and 390% time and space overhead [10, 23]. A number of techniques have been proposed to reduce the instrumentation overhead. Sampling-based techniques, for example, instrument a randomly selected, relatively smaller, subset of program points [3, 23]. A problem with random sampling of program points is that obtaining an adequate profile of program points relevant to a property of interest may require a large amount of samples [15]. Among other things, profiling with respect to a property of interest is helpful for focussing the developer's attention, and for reducing the complexity of profiling results [28].

To illustrate, consider the program shown in Figure 1 which, given a `Order`, processes the order. A property of interest for this example is the use of `userName` in processing orders. If random sampling of program points is applied then a subset of all statements (lines 1–46) will be instrumented and monitored randomly. A majority of elements in these samples are likely to be irrelevant with respect to the property of interest. Thus to adequately cover the statements relevant to the property of interest, an unnecessarily large number of samples would be needed, which will add to the instrumentation overhead. If, on the other hand, the instrumentation overhead is to be kept low, one must settle for a lower coverage with respect to the property of interest.

To increase coverage, while keeping the cost of instrumentation low, we introduce the notion of *property-aware program sampling*. A key insight, which we borrow from Hatcliff *et al.* [14] among others, is to focus the instrumentation efforts on parts of the program that are relevant for a property of interest, i.e. the program slice with the property as slicing criterion [31, 26]. Our profiling technique randomly samples statements from the program slice, as opposed to sampling from all statements in the program as in earlier approaches [3, 23]. For example, in Figure 1, only a subset of program statements will be instrumented, shown as shaded. These statements are relevant to the property of interest. This helps focus instrumentation efforts on the desired parts of a program.

Another issue is that randomly sampling statements provides inadequate coverage of the implicit control relation between sampled statements. These implicit path profiles have potential applications in performance tuning, hot-path prediction, profile-directed compilation, continuous program optimizations and software test coverage with respect to the property under consideration. For example, in Figure 1, the program path that traverses statements on lines 5, 6, 8, 14, 16, 22, 30 is (say) the most likely executed path while processing orders. For e-commerce vendors optimizing this path is of utmost importance, which would require profile information for this path. A random sample of statements may eventually cover all elements on this path, thus giving the 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. The set of slice fragments captures the implicit control structure between the statements of the program slice. In Figure 1, for example, shaded

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.

```
1 public int processOrder(Order order) {
2  int discount = 0; // Irrelevant code elided ...
    ...
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; }
```

Figure 1. An example program and a slice

statements are part of the program slice with respect to the criterion (userName,5). For this slice, an example slice fragment is $\langle entry, 5, 6, 8, 14, 16, 18\rangle$. This slice fragment captures one control structure in the program slice. During sampling, whenever this fragment is selected, there is a higher probability that the profile data for all constituent statements will be collected. This higher probability in turn translates to 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 [30, 32]; 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, which is beneficial for use cases such as feedback-based optimization.

Outline. Section 2 gives the theoretical basis of our approach. The experimental evaluation of a prototype implementation is discussed in Section 3. Section 4 discusses these results, Section 5 compares and contrasts our work with related ideas and Section 6 concludes.

2. Property-aware Sampling

The basic idea behind our approach is that selecting a subset of program entities for monitoring and profiling is likely to reduce instrumentation overhead and facilitate efficient monitoring of a software application [2]. Program slicing produces a subset of program entities that are relevant to a slicing criterion. 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. Others have used this insight for verification tasks but not for guiding instrumentation. For example, Hatcliff *et al.* [14] use program slice for reducing the size of the model that Bandera [18], a model checker for Java verifies. Guo *et al.* [13] use similar technique for limiting the input to their shape analysis technique, etc.

The second insight is that a slice need not be the unit of instrumentation as it often has the tendency to become large [30, 32]. 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 and samples are taken sufficiently often and at random; attaining reasonably accurate profiles at a lower overhead may become possible.

Two possible strategies for decomposing a program slice into parts are possible. Such parts could be formed by randomly selecting a set of statements from the slice and grouping them. An alternative 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 relation are also feasible candidates.

The former approach of decomposition has 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 questions related to individual program points. The profile questions that require implicit path information are harder to answer without significant instrumentation overhead.

The latter approach for decomposing a slice solves this problem. The relationship between statements that constitute a part of the slice facilitates answering path 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. In this work, we only consider control-flow relations for decomposing a program slice.

The rest of this section describes our technique. We first present some necessary terminology. Most definitions are fairly standard and follow from Horwitz *et al.* [16]. Section 2.2 presents the notion of *slice fragments*, a logically-related subset of program slice. Section 2.3 describes the concept of *cover*. Section 2.4 desribes our algorithms for decomposing a program slice into slice fragments.

2.1 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 2.1. *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 2.2. *A* **forward static slice** *$\mathcal{S}$ constructed from program p with respect to criteria $\mathcal{C} = (X, c)$, is the set of statements and predicates that are affected by the values of any variables in X starting at program point c.*

In this paper, any reference to a "slice" or "program slice" refers to a *forward static slice* [27, 26].

For the program in Figure 1, the shaded lines of the code denote statements of forward program slice. The slicing criteria used here was $\mathcal{C} = (\{userName\}, 5)$ i.e. slice criteria for the variable *userName* starting at statement 5 in the program.

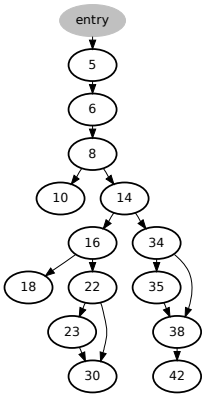

Figure 2. Slice-pruned CFG for program in Figure 1

Definition 2.3. *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*

The pruned graph captures the implicit control flow relationships between the statements of the slice. An example of slice-pruned control flow graph is shown in Figure 2. The computation of this graph is an important step towards *property-aware program sampling*, as it is used to generate the slice fragment population.

Definition 2.4. *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.*

Back-edges in control flow graphs are encountered in case of loops, recursion and return from method invocation.

2.2 Slice Fragments

A crucial part of our approach is to group the program points such that each group – *slice fragment* – is a logically complete set with respect to a property of interest. The following defines this.

Definition 2.5. *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 $\exists v \in \{v_1, \ldots, v_{n-1}\}$ such that $(v_n, v) \in E'$.*

A *slice fragment* captures partial order(s) of implicit control relations between statements in the program slice.

A desirable property of our decomposition, as shown in the following lemma, is that none of the slice fragments can be any larger than the corresponding program slice .

Lemma 2.6. *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 2.5, (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. □

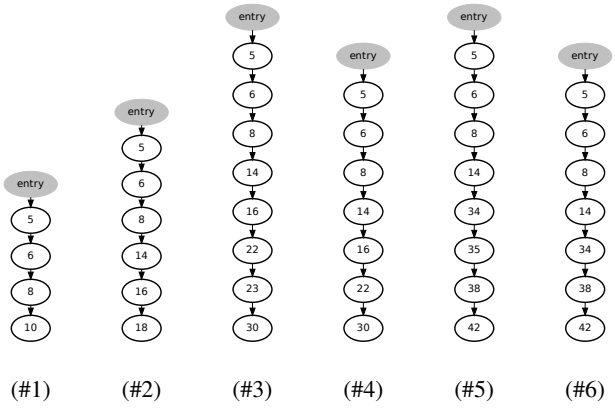

Figure 3. Slice fragments for the slice-pruned control-flow graph shown in Figure 2.

The six slice fragments computed for the graph in Figure 2 are shown in Figure 3. Consider slice fragment #2 in Figure 3: it is a sequence $\langle entry, 5, 6, 8, 14, 16, 18 \rangle$. 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 individual fragments do not contain duplicate vertices.

2.3 Cover and Reduced Cover

The elements of the sample space in our technique are slice fragments. For this sample space to be meaningful, it is required that each statement in the program slice be covered by at least one slice fragment in sample space. More precisely, the slice fragments in the sample space must form a *cover* of the program slice.

Definition 2.7. *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 $v \in \delta$.*

For example, in Figure 3, fragments 1, 2, 3, 4 and 5 together form a cover of the slice shown in Figure 2 because each vertex in the slice is part of at least one of these slice fragment.

A simple way to construct a cover of the program slice is to simply include all the slice fragments. This, however, may cause many of the included slice fragments in a sample to be subsumed by other fragments, which in turn may prove unproductive with regard to a chosen sampling strategy. This motivates the following definition.

Definition 2.8. *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, fragments 1, 2, 3 and 5 together form a reduced cover of the slice depicted in Figure 2. In contrast, fragments 1, 2, 3, 4 and 5 form a cover, but not a reduced cover.

2.4 Program Slice Decomposition

Our sampling approach rests on our technique for decomposing a program slice into slice fragments. Such a program slice can be computed using any of the numerous techniques proposed in the literature (cf. [30]). The discussion of program slice computation is orthogonal to the scope of this paper.

2.4.1 Slice Fragments Computation

The initial step in program slice decomposition is to prune the control flow graph of the program to contain statements only from the slice. This gives us the slice-pruned control flow graph. A depth-first search beginning at the root (criteria) is then performed on this slice pruned control flow graph to generate the slice fragments. During this search we record each root to leaf node sequence as a slice fragment. The back edges encountered in loops, recursion and method return are ignored during the computation. This is to ensure that fragments are acyclic paths. This also implies that a method end point is treated as a *leaf* during the search.

The algorithmic complexity of program slicing depends on various factors (ref. [30, 32]). The time complexity of pruning a control flow graph $G_{cfg} = (V, E, v_0)$ of a program to contain statements from the slice is O($|V| + |E|$). The subsequent depth-first search takes up another O($|V| + |E|$) time to determine the initial raw population of slice fragments.

2.4.2 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.

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 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: **repeat**
6: $\alpha \in \arg\max_{s \in \Theta} |\{v \in s : label(v) = false\}|$
7: Add α to Θ'
8: **for** each vertex v in α **do**
9: Set $label(v) = true$
10: **until** $label(v) = true$ for each $v \in V$
11: **Return** Θ'

Algorithm 1 is in fact a well known heuristic (and approximation algorithm) for the set cover problem [12, 19]. 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 [7]).

Let Θ^* denote the set of slice fragments returned by Algorithm 1. Since Θ itself must be a cover of the slice, and Algorithm 1 does not terminate until all vertices of the slice are covered, we have the following lemma.

Lemma 2.9. *Θ^* is a cover of the slice.*

Proposition 2.1. *Θ^* is a reduced cover of the slice.*

Proof. By Lemma 2.9 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. □

For example, consider the slice fragments in Figure 3. 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. For the first step, slice fragments #3 and #5 are both equally good candidates. Suppose the algorithm chooses 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. In the end, nodes 10 and 18 are the ones not covered so far, resulting in fragment #1 and #2 being picked. It is easy to see that they form a reduced cover of the slice.

2.4.3 Efficient 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.

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)* [24] problem. 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 2.2. *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 $\delta' \in \Theta'$ 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 [11], however, several constant factor approximation algorithms exist for it (cf. [21]). 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 2.2 is monotone. Proposition 2.2 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.

3. Evaluation

To show the feasibility of our technique, we implemented our fragment computation and reduced cover construction algorithms as a stand-alone tool. Our tool uses some functionalities of the IBM T. J. Watson Libraries for Analysis (WALA) [33]. WALA is a static analysis framework for Java bytecode, and provides a rich set of APIs for static analyses.

The objective of the rest of this section is to evaluate the potential utility of our 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. This property is a necessary pre-condition for applying any random sampling technique. The empirical assessment of this property is described in Section 3.2. 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 3.2. 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 3.3.

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. In all the experiments, core Java libraries were excluded from the analysis.

3.1 Subject Programs

For our experiments, we selected a variety of subject programs from different sources. Two programs, namely *nanoxml* and *jmeter* were selected from the software-artifact infrastructure repository (SIR) [17]. *Nanoxml* is a simple SAX parser. *Jmeter* is an application to load test functional behavior and measure performance. We also selected three other open source programs, namely: *jaxen* - a XPath engine for Java, *htmlcleaner* - which transforms HTML to XML, and *xstream* - a library to serialize objects to XML. In addition, *mtrt*, *compress* and *jess* from SPECjvm98 benchmarks were also used. Figure 4 shows some static properties of these programs.

3.2 Assessment of Statistical Significance

We used our prototype tool and a set of slicing criteria to generate slice fragments for the programs mentioned in the previous section. Our tool first computed the entire set of fragments and then applied the reduced cover algorithm (Algorithm 1) discussed in Section 2.4 to create a reduced population of fragments.

Subjects	# of Classes	# of Methods	Bytecode size (in KB)
nanoxml	24	541	35
jaxen	217	1153	389
xstream	331	1519	774
jmeter(core)	242	476	640
htmlcleaner	26	263	79
spec/compress	12	33	18
spec/mtrt	25	470	32
spec/jess	192	1061	67

Figure 4. Static characteristics of subjects

Figure 5 shows the results of the slice decomposition to generate raw population and its subsequent reduction. For each program, the slicing criteria used is shown in the second column. The slicing criterion was selected to be representative of its typical usage.

Analysis. 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. It was observed that program slices of different programs which were almost of the same size, showed huge disparity in corresponding raw populations (*nanoxml/mtrt* and *htmlcleaner/jaxen*), primarily because of different control structures.

The resulting reduced population remained statistically significant, except in the case of *spec/jess*, which showed one of the largest drops in population on application of the reduced cover algorithm. This benchmark applies a set of if-then statements to a set of data. The raw population was low to begin with and most of the branches in the code were subsumed leading to significant drop in population.

The least reduction was observed in *htmlcleaner*. A major portion of *htmlcleaner* is a main controller method that calls a large number of other helper methods as needed. As a result, the control flow of this program demonstrates a large number of acyclic paths out of few nodes that all terminate with JDK calls to write XML file. As a result, no reducible slice fragments are generated. The size of these fragments is a direct reflection of the size of the helper methods that just write the html element and return.

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%). As we discussed previously control flow of *jess* benchmark contains fewer and longer paths. This resulted in longer slice fragments. The *mtrt* benchmark is also similar, where much of the control flow is within a long method `RenderScene` with a number of small branches. We also observed that slice size was determined mostly by the slicing criteria and slice configuration options and not by the size of the program.

3.3 Assessment of Sampling and Coverage

In this section, we empirically evaluate (i) the rate at which the slice fragments are *covered* (selected) under 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, our approach employed a basic sampling technique on the reduced population of slice fragments. We experimented with *simple random sampling without replacement*, whereby a fragment of the population is not chosen more than once in a sample and each fragment had a equal probability of being selected in a sample. This approach is intuitively appropriate as we have minimum advance information about the population of slice fragments. For select programs, the results of (i) and (iii) are presented in Figure 6. The numbers beside the sample sizes in the labels indicate the average number of unique statements for a corresponding sample.

Subjects	Slicing criteria	Slice size (S)	Raw Slice Fragments			Reduced Slice Fragments			
			POP	Fragment len.(avg)	Fragment len.(% of S)	POP	Reduction (% of Raw POP)	Fragment len.(avg)	Fragment len.(% of S)
nanoxml	<XML> to be processed	140	1417	43	30.71	97	93.15	42	30
jaxen	XPath expression	232	79	12	5.17	50	41.42	13	5.60
xstream	<Object> for XML conversion	173	9482	31	17.91	322	96.6	26	15.02
jmeter	<File> to the core utility	57	48	10	17	34	29.2	10	17.5
htmlcleaner	<File> to clean	265	295	26	9.8	292	1.02	26	9.8
spec/compress	Arg. to *benchmark*	397	3267	94	23.67	46	98.59	79	19.899
spec/mtrt	Arg. to *benchmark*	130	269	50	38.46	37	86.24	56	43.07
spec/jess	Arg. to *benchmark*	42	53	15	35.71	13	71.69	15	35.71

Figure 5. Slice decomposition and reduction algorithm on subject programs: for each subject, the size of the computed slice (*S-number WALA statements*) and characteristics of the generated slice fragment populations(*POP*) are tabulated.

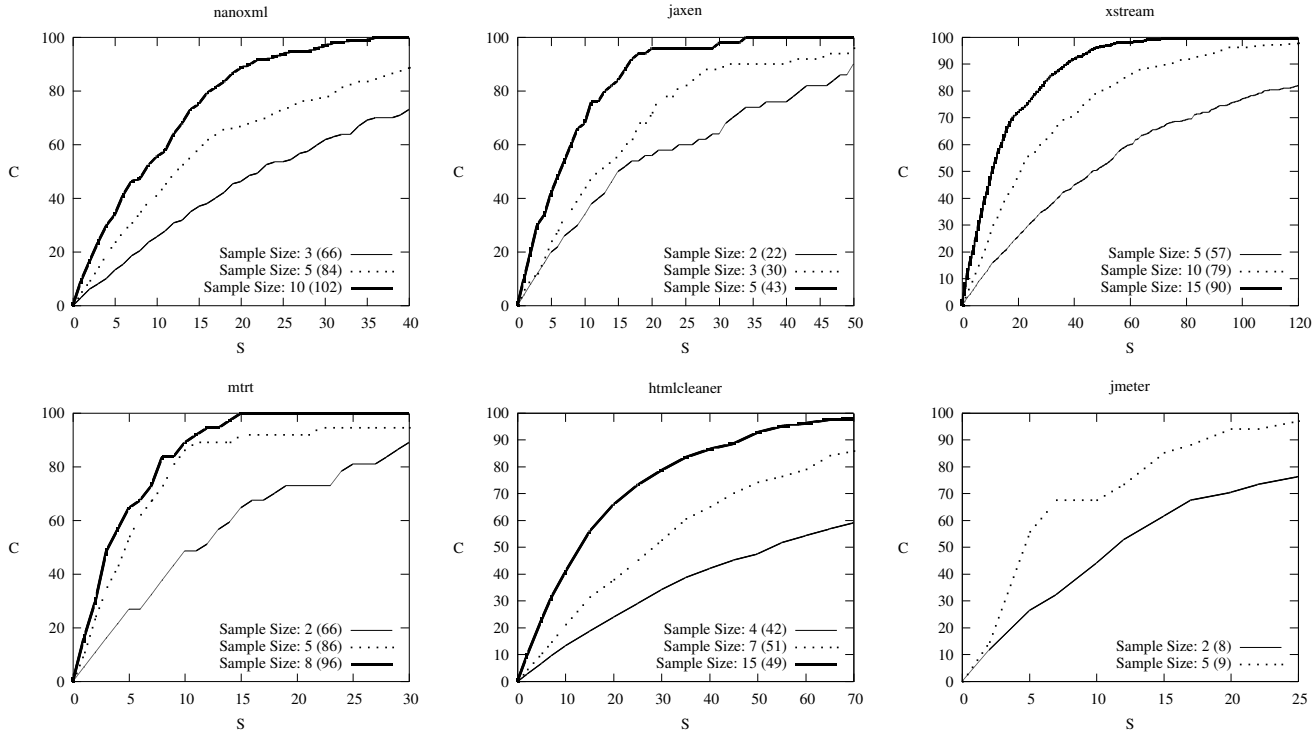

Figure 6. Statistical Analysis of Slice Fragment Coverage: Fragment Coverage(C as %) vs Number of Samples(S)

The experiments were repeated for a number of different sample sizes. The range of sample sizes that we experimented with was proportional to the population of slice fragments, but we only show representative set of the sample data for the subject programs. Figure 7 depicts the corresponding details on (ii).

Analysis. 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. The results were in line with this intuition. This disparity was most pronounced in case of *xmlstream*. It showed that under different sample sizes, sampling 25 times was sufficient to reach near optimal statement coverage, whereas 150 samples were required to observe the same in case of slice fragments. *jmeter* and *mtrt* showed skewed fragment coverage, due to small population. The average number of unique statements per sample (see label of Figure 6) was observed to be significantly smaller than the corresponding program slice size.

4. Discussion

Our experimental results showed that the population of slice fragments were statistically significant in almost all cases. These results offer preliminary evidence that our technique is likely to produce statistically significant population for large slices, which is essential for sampling.

The slice fragments are units of monitoring tasks. Our results showed that their average size was small in most cases, thus our approach is likely to reduce the number of instrumented points.

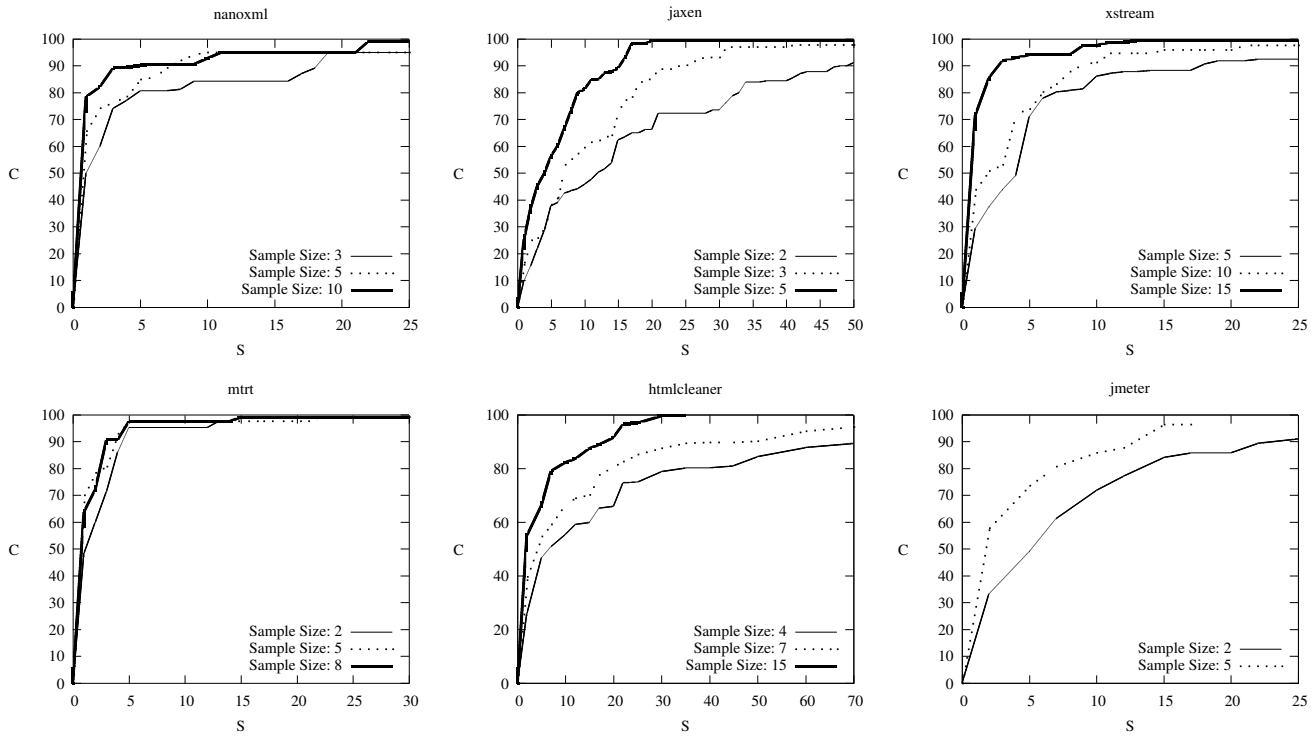

Figure 7. Statistical Analysis of Statement Coverage: Statement Coverage (C as %) vs Number of Samples(S)

Our study of random sampling of slice fragments showed that the number of iterations required for covering all the slice fragments in the population depended largely on the sample size. These results show that if the samples are taken frequently, the fragments in the population will be covered often for the profile to be useful.

In addition, the average number of unique slice statements per sample was found to be a small fraction of slice. This implies that on average only a small fraction of slice is likely to be instrumented when a sample is taken, leading to low monitoring overhead.

Scalability. Our experiments demonstrate that our approach is more likely to be beneficial in cases where the program slice size is large, resulting in a statistically significant population after decomposition and reduction. Fortunately, this is precisely the scenario where an approach for reducing the scope of instrumentation would be needed. If sample sizes are chosen prudently with respect to the slice fragment population, then the average number of statements instrumented per sample is likely to be low, resulting in scalable monitoring processes

Comparison with random sampling of statements. A process based on random sampling of statements in the entire program would have a significantly lower probability of capturing implicit control paths that our technique captures. This is because a random selection of statements is not constrained to be along partial paths (fragments). Thus we didn't find it necessary to do a comparative empirical evalutation against this technique as it would have simply served to validate the obvious.

Runtime overhead. We have focused on what parts of the program to instrument. The issue of how to instrument was not considered, which to a large extent would determine the actual runtime overhead. "How to instrument" is an active topic with several results that are complementary to our approach. Nonetheless, we procured prelimnary results on a naive instrumentation technique we experimented with. Over a period of sampling iterations, for a set of inputs across different subject programs, we observed that the slice fragments ended up getting *captured* 15–60% of the time (which lead to more relevant monitoring information). Also, the runtime overhead due to the instrumentation in our approach was found to be 35–94% lower when compared to full-slice instrumented version of the program.

5. Related Work

In this work, we have proposed selecting a subset of program slice entities for monitoring and profiling software, using statistical sampling schemes. This section discusses closely related ideas.

We share a similar objective of reduction in monitoring overhead with Santelices *et al.* [28], which proposes a subsumption algorithm based on the *type* of control-flow entities and Apiwattanapong *et al.* [2], which proposes a method to monitor selective paths of a program. Our approach differs in that it uses program slicing and sampling to attain similar results.

Arnold and Ryder [3] 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.

We share a similar goal with the GAMMA system [5, 25]; that is, a reduction of monitoring overhead in deployed software instances. Their main idea is to divide and allot the monitoring tasks across several instances of the software. They then collect the data from these instances to compute monitoring information for the complete application. MOP [6] is a runtime verification framework, which generates monitors from the specified properties

and integrates it with the application. In contrast, we apply slicing and sampling to reduce the scope of monitoring.

Thin Slicing [29] proposes a selective notion of relevance based on a seed computation to reduce the scope of debugging and program understanding tasks. The application of thin slicing is geared towards debugging and program understanding tasks, whereas our approach is oriented towards profiling and monitoring software.

Liblit *et al.* [22] 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. We use sampling to control the amount of instrumentation over the relevant set of program points.

Dwyer *et al.* [9] 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 property-relevant fragments for profiling.

6. Conclusion and Future Work

The key technical contributions of this work are: (i) the notion of slice fragments, (ii) a method to compute them, and (iii) 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 these slice fragments is then instrumented for monitoring or profiling tasks. We also presented empirical results to validate that our technique has the capability to collect relevant profiles, at a significantly lower overhead.

Several interesting avenues remain to be explored. An empirical study could be conducted to find class of programs, where our approach can be applied to guide an existing monitoring technique. 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.

Acknowledgement. This work was supported in part by the US NSF under grant CNS-06-27354 and by CNS-08-08913.

References

[1] 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.

[2] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *PASTE '02*, pages 35–42, 2002.

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

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

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

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

[7] 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.

[8] S. Debray and W. Evans. Profile-guided code compression. In *PLDI '02*, pages 95–105, 2002.

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

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

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, Jan. '79.

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

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

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

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

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, 1990.

[17] 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, '05.

[18] James C. Corbett *et al.*. Bandera: extracting finite-state models from Java source code. In *ICSE '00*, pages 439–448.

[19] 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.

[20] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02*, pages 467–477.

[21] S. Khuller, B. Raghavachari, and N. Young. Approximating the minimum equivalent digraph. *SIAM J. Comput.*, 24(4):859–872, '95.

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

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

[24] 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.

[25] 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.

[26] 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.

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

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

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

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

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

[32] 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.

[33] T.J. Watson libraries for analysis. `http://wala.sourceforge.net`.