Offline Framework for Performance Comparison of Software Revisions

Nagy Mostafa and Chandra Krintz
Computer Science Department
Univ. of California, Santa Barbara
{nagy,ckrintz}@cs.ucsb.edu
UCSB Technical Report #2008-19 Nov. 21, 2008

Abstract

Repository-based version control systems such as CVS, RCS, Subversion, and GIT, are extremely useful tools that enable software developers to concurrently modify source code, manage conflicting changes, and commit updates as new revisions. Such systems facilitate collaboration with and concurrent contribution to shared source code by large developer bases. In this work, we investigate a framework that lays the groundwork for "performance-aware" repository and revision control for Java programs. Our framework automatically tracks behavioral differences across revisions to provide developers with feedback as to how their change impacts performance of the application. It does so as part of the repository commit process by profiling the performance of the program or component, and performing automatic analyses that identify differences in the dynamic behavior or performance between two code revisions.

In this paper, we present our analysis system that is based upon and extends prior work on calling context tree (CCT) profiling and performance differencing. Our framework couples the use of precise CCT information annotated with performance metrics and call-site information, with a simple tree comparison technique and novel heuristics that together target the cause of performance differences between code revisions without knowledge of program semantics. We evaluate the efficacy of the framework using a number of open source Java applications and present a case study in which we use the framework to distinguish two revisions of the popular FindBugs application.

1. Introduction

Software developers world-wide employ revision control (RC) systems for managing a vast number and diversity of open-source and proprietary software code bases. RC systems facilitate and support distributed, collaborative, and incremental contribution to shared source code via storage repositories and a set of tools that provide, among other things, access to files, management, tracking, and branching of revisions, automatic resolution of conflicts, and feedback to developers when automatic conflict resolution fails or when events occur by other developers.

Client-server RC systems include CVS, RCS, Subversion, and the Visual Studio Team System (VSTS); popular RC systems that implement distributed local repositories include GIT, Fossil, Mercurial, and Codeville.

Although these RC systems are stand-alone applications (the focus of our work), support for revision control can be and is integrated into other applications such as word processors, spreadsheets, and databases. RC systems enable developers to access and branch off of earlier revisions of a project or file and in some cases provide other services such as automatic testing (Visual Studio Team System (VSTS)) and defect or issue tracking (e.g. Codeville, Fossil, VSTS).

In this work, we are interested in providing a new service for RC systems: tracking of revision performance and dynamic behavior differences. To enable this, we have designed and implemented *performance-aware revision control support (PARCS)*, a service that provides feedback to developers as to how a change that they have committed affects the behavior and performance of the overall application. Given the complexity of hardware and software and the popularity of collaborative development, such tools are key to helping developers understand the behavior of large applications and how local and incremental modifications impact overall performance over time.

PARCS is a program profiling and analysis framework that executes a program using test inputs when a new source code revision is checked into an RC repository. PARCS builds upon and extends prior work on calling context tree (CCT) profiling [2, 14, 6, 4, 17] and performance differencing [16], but is unique in that it targets two different versions of the same program with the same input on the same platform. Prior work has focused on identifying performance differences across two executions of the same program using different inputs or underlying platforms [16]. PARCS instruments the program and generates a precise CCT during offline (background) execution. PARCS annotates the CCT with a number of different performance metrics and can store CCTs for later comparison across revisions.

To find CCT differences, PARCS first transforms the trees using common tree matching that we extend with feedback from changes that the developer has made to the code and simple relaxation techniques. As a result, PARCS incrementally identifies topological differences in the CCTs of two revisions. PARCS then classifies these differences into four categories that distinguish the reason behind the performance differences: method addition/deletion, direct code modification, indirect code modification effect, and non-determinism. PARCS excises all subtrees rooted at nodes where these differences originate.

The CCTs that result after this pruning are identical topologically. PARCS analyzes these trees for differences in the performance metrics that annotate the trees. For this step, PARCs employs simple weight matching and performs an iterative algorithm to identify pairs of nodes with weight differences that are significant, i.e., that are

larger than the differences that are typical of a non-deterministic effect. Finally, PARCS attributes the topological and weight differences to specific code changes and reports its findings back to the developer.

We implement a PARCS prototype for Java programs via extensions to the OpenJDK JVM from Sun Microsystems. We empirically evaluate the efficacy of PARCS using a number of open source Java programs and employ PARCS to identify differences between two revisions of these applications. Although the test input that PARCS employs for CCT creation determines the amount of topological and weight difference that it discovers, we find that a single, well chosen input can lend significant insight into behavioral and performance differences that result from source code changes. PARCS however, is able to investigate any number of inputs concurrently. We describe a detailed case study that we perform to attribute behavioral and performance differences to changes made between revisions using a single input for the popular FindBugs application [10]. Overall, we find that PARCS is easy to use and highly effective at helping to identify the cause of revision-based behavioral and performance differences.

In the section that follows, we provide background on the key components that PARCS builds upon and extends. We then detail the PARCS design and implementation (Section 3). In Section 4, we present a case study on our use of PARCS for revisions of the popular FindBugs application. In Section 5, we empirically evaluate PARCS using a number of different open source applications. We then discuss related work (Section 6) and present our conclusions and plans for future work (Section 7).

2. PARCS

PARCS is performance-aware repository control support that identifies dynamic behavioral and performance differences that result from changes to source code from revision to revision. To enable this, PARCS employs a dynamic calling context tree (CCT) for collection and evaluation of dynamic program behavior. PARCS compares two software revisions by identifying the topological and weight-based differences between the CCTs of the two revisions. We overview the background on CCTs and CCT topological differencing in the subsections that follow. We detail the PARCS implementation of weight-based differencing in Section 3.

2.1 PARCS Dynamic Program Representation and Collection

PARCS collects, manipulates, and compares the dynamic behavior of a program using a data structure called a calling context tree (CCT) [2]. A CCT captures the calling context of each dynamic method invocation that occurs during program execution. Given a method call stack, the calling context is the list of methods that are invoked without returning to reach a particular program point (in this case a method call).

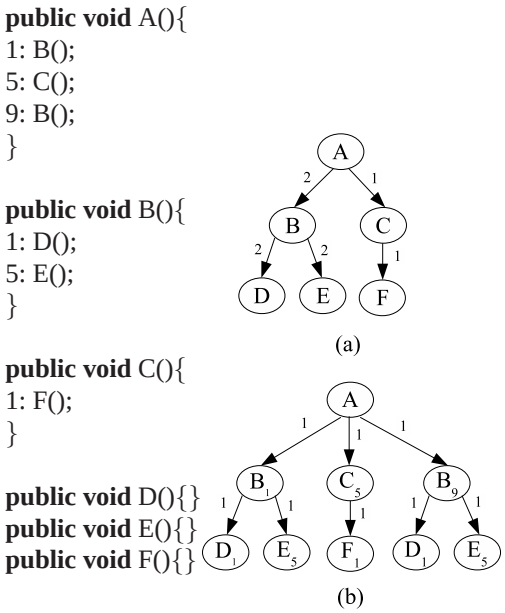


Figure 1. Code snippet with the corresponding CCT. (a) shows the corresponding CCT with no call-site information included. (b) shows the equivalent tree with call-site information shown as subscript

A CCT summarizes the dynamic call tree (DCT) of a program. Each node in the DCT is a single activation of a method during execution. An DCT edge from node X to node Y, represents a dynamic invocation of Y from the code in X. DCTs can be annotated with additional runtime information such as execution time and arguments for each node. One way to summarize the DCT, to reduce its potentially enormous size, is to instead use a dynamic call graph (DCG). In a DCG each node corresponds to *all* activations of a method. Such a summary however can lose significant amounts of behavioral information about the execution of a program.

The CCT is a data structure that provides a middle ground – it does not require the space of a DCT but captures important behavior information that a DCG loses. In a CCT, all activations of the same method that execute from the same calling context are aggregated into a single node. An edge from node X to Y, as in the DCT, represents a call from X to Y. The calling context of a node Y, thus, is captured by a series of nodes from the root of the tree to node Y. The edge between two nodes records the number of times the program executes that particular calling context.

Figure 1 illustrates an example of a CCT for a program with methods A through F. Assuming that A is the entry method, (a) shows the CCT for a particular execution of the program. The numbers on the edges are invocation counts. For example, the invocation count on the edge B→D is 2, which means that D is called twice from the context A→B.

PARCS employs CCTs for its profile collection. However, we extend its implementation to distinguish call-sites (prior work considers all calls to a method Y within method X to have the same context [16]). In our CCTs, PARCS records a method Y that is called from two different call-sites within method X independently from each other. Figure 1 (b) shows the CCT with call-site information (shown as subscripts). Distinguishing based on call-site information increases the size of the CCT but provides more details about the execution that are useful to developers for identifying behavioral and performance differences across revisions. The authors in [7] show that call-site information is also important for coverage testing and anomaly detection using CCTs. We evaluate the trade-off between size and accuracy of employing call-site information for identifying performance differences in Section 5.

The call-site CCT serves as a suitable data structure for comparing performance across program revisions as it captures context information which helps programmers better understand performance and correlate it to the program semantics. Context information expressed as stack traces are still the most widely used means of describing program point of failure. Moreover, CCTs provide a good trade-off between size and accuracy compared to DCTs and DCGs.

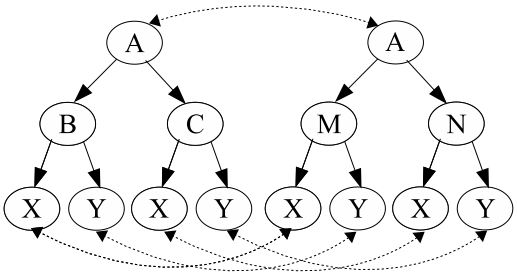
PARCS instruments each method entry and exit of the program to collect the CCT. Since PARCS is employed by a revision control system off-line (in the background), we do not consider the overhead of exhaustive profiling of the calling contexts. Exhaustive profiling is important for PARCS since it is able to capture all calling context behavior. PARCS annotates the collected CCTs with other profile information such as execution time and invocation count. The PARCS framework is extensible enabling researchers to investigate its efficacy using other performance metrics.

2.2 Identifying Topological Differences in CCTs

PARCS compares two CCTs to identify the topological differences between them. In the subsections that follow, we consider two well known topological tree matching algorithms: tree transformation and common tree matching. We then present relaxed common tree matching, the algorithm that PARCS employs to identify topological differences.

2.2.1 Tree Transformation

Shasha et al. [1] propose a tree comparison algorithm for ordered trees; they employ dynamic programming for its implementation. An ordered tree is a tree in which the children of each node have total order. Given two trees, the algorithm finds a sequence of operations that, when applied, transforms one tree to the other. The algorithm



Operations: (1) Rename C to N, (2) Rename B to M

Figure 2. Example of tree transformation.

is optimal in the number of transformation operations used (the edit distance) and has a time complexity of $O(|CCT1| \times |CCT2|)$. The primary tree operations used are:

1. *Delete X:* delete node X and move its children to its parent Y; the children are inserted at the same position in the child order of Y at which X was positioned.
2. *Insert X, Y, P:* add node X to be a child of node Y at position P in the children order of Y. X gets a consecutive subsequence of Y's children.
3. *Rename X, Y:* rename node X to Y.

Although this algorithm was originally designed for abstract trees, Zhuang et. al employ the algorithm to compare two CCTs for the *same* program that they execute on different platforms or with different inputs [16]. The authors in this prior work use the number of operations required to transform CCT1 to be CCT2, as a *difference metric* with which they compare two trees.

Figure 2 shows two CCTs with topological difference and the sequence of resulting operations that transform the left CCT to the right CCT. After applying the transformation the two CCTs become identical. All nodes that are not involved in any transformations are matched nodes. The dotted arrows in Figure 2 shows the matching.

The way the algorithm matches nodes relies solely on the node label and its post-order in the tree. It ignores the context of the node (path from root to the node) and hence may match nodes with the same method name but different contexts. For example, in Figure 2 method X called by method C may have completely different semantic than X by method N. Considering the two to be equivalent could be misleading to performance analysts. Since in PARCS we are comparing two versions of a program, these inaccuracies are more likely to occur more often since the code is different across versions. Inaccuracies in PARCS lead to incorrect identification of differences and attribution of differences to code changes.

Also, using dynamic programming incurs quadratic time and space overhead. While this is tolerable for small CCTs, it becomes hindering for larger ones. Since we rely on call-site CCTs for more accurate differencing, using this algorithm becomes infeasible. For example, the call-site CCT of FindBugs has 155,787 nodes. Thus an array of more than 22.5 billion entry is needed.

For the above reasons, we investigate an alternative approach to CCT matching based on common tree matching.

2.2.2 Common Tree Matching

Common tree matching is a well-known, simple technique for comparing two trees. The algorithm traverses the tree level-by-level, comparing nodes. Each node in the tree has an order. The order of node n is the position of n amongst its siblings. For example, in Figure 3 (a), the order of nodes A, B and F are 1, 1 and 2, respectively.

We define equivalence of two nodes recursively as follows:

Definition 1. Node Equivalence:
Node $n_1 \in CCT_1$ is equivalent to node $n_2 \in CCT_2$ iff

1. n_1 and n_2 have the same method name
2. n_1 and n_2 have the same order
3. n_1 and n_2 have equivalent parent nodes

This definition implies that equivalent nodes always have the same context. Moreover, if two nodes are not equivalent, we consider the subtrees rooted at them a topological difference.

Figure 3 (a) illustrates a common tree matching example. First, we compare root nodes, since they are equal, we proceed to the second level (A's children). On the second level, the first node B exists in both trees, thus we consider it on the common tree and will process all of its children once we move to the third level. The second node C in the left tree corresponds to F in the right, which is a mismatch; we report both C and F and their subtrees as a topological difference. We do the same thing (apply a mismatch) for node C in the right tree. The grey nodes constitute the resulting common-tree.

The problem with common-tree matching is that it follows a very conservative definition of equivalence. In the right tree of Figure 3 (a), method A has been modified to call method F before it calls C. If this is the case, we should report C as part of the common-tree. Because of the definition of equivalence, we report C in the right tree as a mismatch. To overcome this limitation and to capture such changes to source code, we relax the

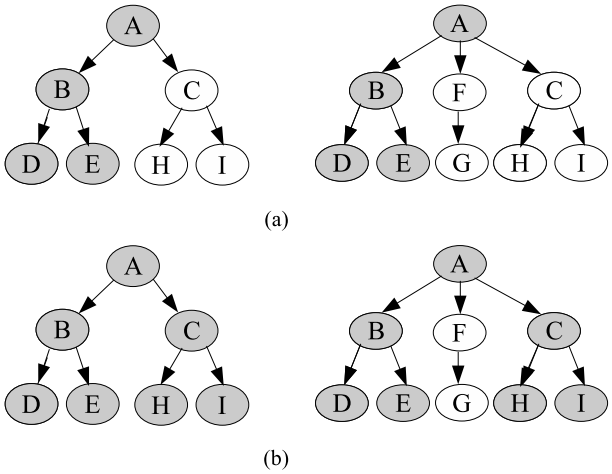


Figure 3. Common tree matching example. (a) shows strict common tree matching. (b) shows the relaxed common tree matching. Grey nodes are the common tree found in each case

definition above to use the relative ordering among matched nodes instead. Our definition of equivalence then becomes:

Definition 2. Relaxed Node Equivalence:

Node $n_1 \in CCT_1$ is equivalent to node $n_2 \in CCT_2$ iff

1. n_1 and n_2 have the same method name
2. n_1 and n_2 have equivalent parent nodes
3. all matched left siblings of n_1 are matched to left siblings of n_2 and all matched left siblings of n_2 are matched to left siblings of n_1.

We refer to the version of the algorithm that employs this definition as *relaxed common-tree matching*. Using this algorithm, equivalent nodes still have the same context. The difference is that they do not have the exact child order. Figure 3 (b) shows that using Definition 2, we can match node C. We employ relaxed common-tree matching within PARCS to identify topological differences between two CCTs (program revisions).

3. PARCS Implementation

Figure 4 overviews the PARCS process. We employ PARCS for revisions of Java programs in our current prototype. We start by checking out the source code of the two revisions of interest from a code repository (e.g. CVS). We then compile the source code to bytecode. Next, we run the two revisions using the same test input via a modified Java Virtual Machine that builds CCTs from the execution. We can generate the CCTs of earlier revisions on-the-fly, in parallel, or store them in the repository. Developers can specify the input that PARCS uses to generate CCTs; PARCS can evaluate multiple inputs and CCT pairs concurrently.

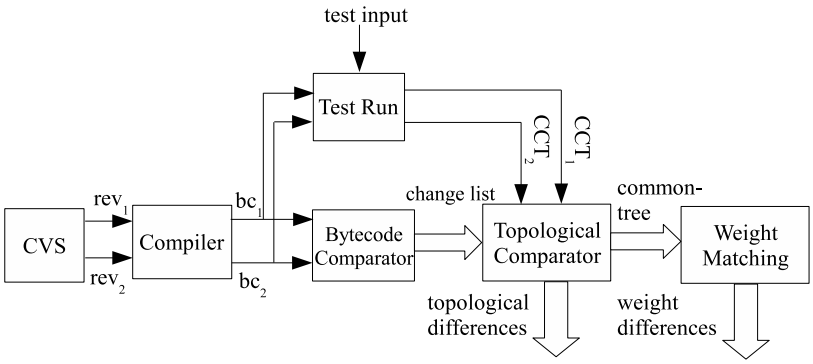


Figure 4. Framework overview. rev_1 and rev_2 refer to old and new revisions, respectively. bc_1 and bc_2 are the corresponding bytecode.

In addition, PARCS performs a fast, static bytecode comparison on the revisions to extract method-level changes. PARCS feeds the CCTs and this bytecode change-list into the incremental topological comparator. After this component removes all topological differences from the CCTs, PARCS performs weight matching on the resulting trees to identify nodes with the largest differences in performance metrics. We detail each of these steps in the following subsections.

3.1 CCT Collection

The PARCS system generates CCTs by exhaustively recording all application methods calls and returns. For this study, we record only application methods and ignore calls to the Java runtime and library code to keep CCT sizes small and CCT processing fast. We can easily extend this system to include library calls if necessary. Since the libraries will not change from revision to revision, we need only perform weight matching on them to identify performance differences that occur indirectly from application code changes. Our CCTs, as described earlier, distinguish contexts for each call-site invoked. The performance metrics with which we annotate CCTs include invocation count, and the average and standard deviation of execution time. We store all CCTs in a relational database for future analysis.

3.2 Method-level Bytecode Comparison

After compiling the two revisions, we perform a bytecode comparison to generate a list of all added, deleted, modified, and renamed methods. The process starts by compiling source files from each revision code base to get the set of class files. The class files therefore belong to either the application or any local Java modules it uses. We do not consider class files that are dynamically downloaded over a network or created at runtime.

We have chosen to implement this comparison on the Java virtual machine intermediate representation (bytecode) rather than source code because of its compact and readily available format as opposed to manipulating

the diff files of repository. Also, some source code changes are useless to PARCS since they have no effect on the program semantics (e.g. variable declaration relocation within a method, variable renaming, replacing a for-loop with a while-loop, ... etc.). Most of these changes are not reflected on the bytecode and hence are automatically ignored. The same argument holds for any other virtual machine intermediate form.

We match the class files from the two revisions according to their package and class names. For each matched pair of class files, we generate a list of methods that each class file contains. By comparing the two lists, we build the following method sets:

1. *Added Methods:* methods present in the new revision but not in the old one.
2. *Deleted Methods:* methods present in the old revision but not in the new one.
3. *Modified Methods:* methods present in both revisions with everything identical except for the code body.
4. *Renamed Methods:* methods present in both revision with everything identical except for the method name.

We compare methods by their fully qualified names and code bodies. A fully qualified method name consists of the full package name, class name, and method signature. The method signature consists of method name, number and type of parameters, and return type. We consider a method modified, if only the code body has been changed. Renamed methods have only changed method names.

3.3 Incremental Topological Comparison

There are four primary reasons for topological differences between two revisions:

- **Reason 1** *Addition/Deletion of Methods:* any calls to such methods introduces a topological difference.
- **Reason 2** *Direct Modification:* code modification that explicitly enables/disables or adds/deletes a call to a method existing in both revisions.
- **Reason 3** *Indirect Modification:* a change in the program that has a side effect. For example, a global variable update or a configuration file change that affects which methods are called. Also, some modifications may have hidden effect on another method execution time, such as the effect of cache thrashing.
- **Reason 4** *Non-determinism:* randomness in the program execution.

Using the code change information that we obtain from the bytecode comparison, we label CCT nodes as “added”, “deleted”, “modified” or “renamed”. This mapping of code change to the dynamic CCT enables topological differencing to proceed incrementally. Using these reason categories, we apply the relaxed common-tree matching technique that we describe in Section 2.2.2, incrementally in three stages:

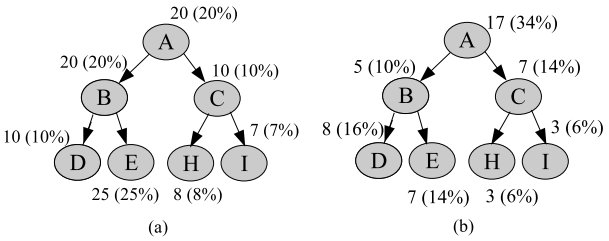


Figure 5. Weight matching example. Common trees with identical topology and different weights.

Stage 1. We excise all subtrees rooted at added and deleted nodes from the CCTs and log each change for later attribution (Reason 1).

Stage 2. We identify topological differences that are most likely due to direct modification (Reason 2). Given the set of modified nodes, we identify the modified nodes that are highest dominators in the tree. X is a dominator of Y, if the path from the CCT root to Y contains X (and X and Y are both modified nodes). A highest dominator is a modified node with no modified dominators (i.e. highest in the tree). Using this definition, we match highest dominators across the two CCTs using method signatures and contexts. We ignore unmatched nodes as we handle them in Stage 3. For each pair of matched highest dominators, we perform relaxed common-tree matching on the subtrees rooted at them. We report all differences that we find as potentially resulting from direct modification. Although this is the most likely cause (and is the most common in our experience), it is possible that the differences we identify result from side-effects or non-determinism. We excise these subtrees and report each change.

Stage 3. Finally, we conduct a global topological comparison for what remains of the two CCTs and record each change. These topological differences are either due to side effects or non-determinism (Reason 3 and 4). We excise (and report) all subtrees rooted at unmatched nodes.

3.4 Identifying Weight Differences

With all topological differences reported and omitted from the two CCTs, the parts remaining are identical in topology but they may vary in performance metrics. PARCS next performs weight matching to identify the differences in weights across CCTs. Weight differencing is a key component of PARCS since it identifies differences that are due to changes to the code made by the developers that change functionality without changing the method call behavior. In addition, weight matching identifies behavioral and performance differences due to modification side effects (and non-determinism).

The PARCS weight matching algorithm quantifies the degree of similarity between the two trees in terms of their annotated performance data using an overlap metric defined and used in prior work [9, 5, 6, 16]. We define

overlap in our setting as:

$$overlap(|CCT_1|, |CCT_2|) =$$

$$\sum_{\substack{n_1 \in CCT_1 \\ n_2 \in CCT_2 \\ n1 \equiv n2}} min(pweight(n_1, CCT_1), pweight(n_2, CCT_2))$$

where $n_1 \equiv n_2$ means that n_1 in CCT_1 is equivalent to n_2 in CCT_2 (the two nodes match). We define $pweight(n_i, CCT_i)$ as the percentage of the total weight across all nodes in CCT_i that is represented by the weight of node n_i in CCT_i. The degree of overlap ranges from 0% to 100% and indicates how much of the performance of CCT_1 is similar to that of CCT_2, i.e. how much of CCT_2's performance is covered by CCT_1. 100% overlap indicates perfectly identical CCTs. Note that since there is non-determinism and noise in performance data – it is likely to be the case that two CCTs generated by two different runs of the same program on the same platform with the same input, do not have 100% overlap. For example, the latest revision of the popular FindBugs application (that we analyse in the next section), has a 99.3% overlap in execution time using our test input. Figure 5 illustrates the common-trees from Figure 3 that PARCS has annotated with absolute node weights and $pweights$ (shown in parenthesis). The overlap of the two CCTs is 76%.

To identify the pairs of nodes that constitute the most significant performance difference, we employ this overlap metric as part of an iterative weight matching algorithm based upon that employed in [16] for node matching. Our algorithm is parameterized by an overlap threshold (and number of nodes of interest). However, we automate generation of the overlap threshold value by computing the overlap percentage of two CCTs for the latest revision – the same program, on the same platform, using the same input, that we execute twice. This overlap value captures the difference that we expect from noise and non-determinism. Developers can set this threshold to a different value, to investigate other weight difference pairs, if so desired. Alternatively, developers can specify the number of nodes they are interested in investigating, i.e., it is possible to specify that no more than 10 matched nodes are returned. The nodes that PARCS returns are the methods responsible for the greatest contribution to the overall weight difference between the two revisions.

We first compute the $pweight$ difference between each pair of matched nodes. We sort the pairs of nodes in decreasing order of this difference. The pair sequence for the example in Figure 5 is A, E, D, C, H, B and I. We compute the overlap of the CCTs and compare it to the threshold. If the overlap value is less than the threshold, we perform an iterative computation to adjust the weights (to produce a new overlap value).

Specifically, we iterate over the pairs in order of largest to smallest $pweight$ difference, and for each pair we adjust the small $pweight$ to be equal to the large one. We then recompute overlap and compare this value to the overlap threshold (or to the maximum node count specified if any). Once either threshold is reached, PARCS reports the nodes for which it performed weight adjustment.

In our example, we first adjust A's weight in the right CCT to be 8.25 (which is a $pweight$ of 20% to match that of A in the left CCT). When we recompute overlap, the result is $\approx$ 83.35%. We compare this resulting value to the overlap percentage and the number 1 (since we have adjusted one pair) to the specified maximum number of nodes to return, if any. We continue to adjust the second pair (E), if neither one of these constraints is met.

3.5 Attributing Topological and Weight Differences

In our current prototype of PARCS, we report each difference with an ordered list of methods that are most likely to be the cause of the difference. We also report supporting evidence and data for each method (context, performance metrics, etc.). In addition, PARCS presents the context information (annotated subtrees, complete CCTs with highlighted node differences, etc.) to developers in graphical format for easy viewing and investigation. The source repository software identifies all changes made to each of these methods that we report. The exact attribution of a difference to a specific change however, proceeds by hand – however with PARCS support (described below). We walk through an example of this process in the next section for two revisions of the FindBugs application.

To identify the most likely methods causing each difference that PARCS identifies, we employ a simple heuristic. We consider the reasons we list in Section 3.3 as the potential causes. For Stage 1 differences, we report the parents of the excised subtrees (callers to added/deleted methods). For subtree excised in Stage 2, we report the list of modified nodes on the path from the subtree root to the CCT root starting from the closest modified dominator upwards. For Stage 3 and weight matching, we report the differences along with their context.

4. Usage Example: FindBugs

In this section we demonstrate, by example, how we apply these heuristics to identify the reason for topological and weight differences. To enable this, we compare the CCTs of two revisions of FindBugs [10], a Java tool to find bugs statically in Java code.

First, we execute Stage 1 of the algorithm to remove all subtrees rooted at added/deleted nodes and Stage 2 to find subtrees of differences dominated by modified nodes. Figure 6 visualizes a subset of the CCT from the

latest FindBugs revision. We only show node ID's for convenience and we identify the modified methods with rectangular nodes. The nodes in gray are those that Stage 2 identifies as different from the CCT of the earlier revision. These are the nodes that Stage 2 removes. Stage 2 returns the list of all modified nodes between the subtree root and the CCT root for all removed subtrees. PARCS orders the list from the modified node nearest to the subtree to the furthest. For the subtree rooted at node 29747, PARCS returns the list {29744, 19913}.

For this case study, we first investigate the reason behind the topological differences in the three subtrees rooted at 29745, 29747 and 29748 which correspond to methods:
Item.init(), Item.makeCrossMethod() and *Item.equals()*, in the FindBugs application, respectively. As we describe previously, there are three potential reasons behind these differences. The first and most likely reason is direct code modification that introduced/enabled these calls. In such cases, the modified method nodes will be one of the ancestors of the subtrees roots (in this case nodes 29744 and 19913) that Stage 2 returns. The second reason is a side effect of some modification that indirectly causes the subtrees. Finally, the reason may be non-determinism during execution.

We begin by investigating the methods that correspond to the nodes that Stage 2 reports (29744 then 19913): *FieldSummary.setComplete()* and *FindBugs2.analyzeApplication*, respectively. Using the differences in the source code of these methods reported by the source code repository (or our bytecode analysis tool), we find that the modified method *FieldSummary.setComplete()*, inserts these three calls in the latest revision but not in the former.

We repeat the same procedure to find the cause for the different subtrees rooted at nodes 130194 and 130190. The ordered list of candidate methods that Stage 2 reports is {78182, 19913}. Again, we start by the node closest to the subtree root which corresponds to method *FindUnrelatedTypesInGenericContainer.analyzeMethod()* which implements a source code change that inserts the two calls.

After removing all topological difference during Stage one and two, the only topological differences remaining, if any, will be due to either indirect modification or non-determinism of execution. By running stage three of the algorithm, we find one tree removed from each CCT both rooted at method *JavaVersion.clinit* (not shown in the figure). This method is the class initializer for the class *JavaVersion*. Analyzing this method, we find that non-determinism is the reason. In particular, the use of the Java data structure *HashSet* makes no guarantee to the iteration order of the set. The order by which the items in the *HashSet* are processed dictates the point at which the class initializer of *JavaVersion* is invoked to cause topological difference.

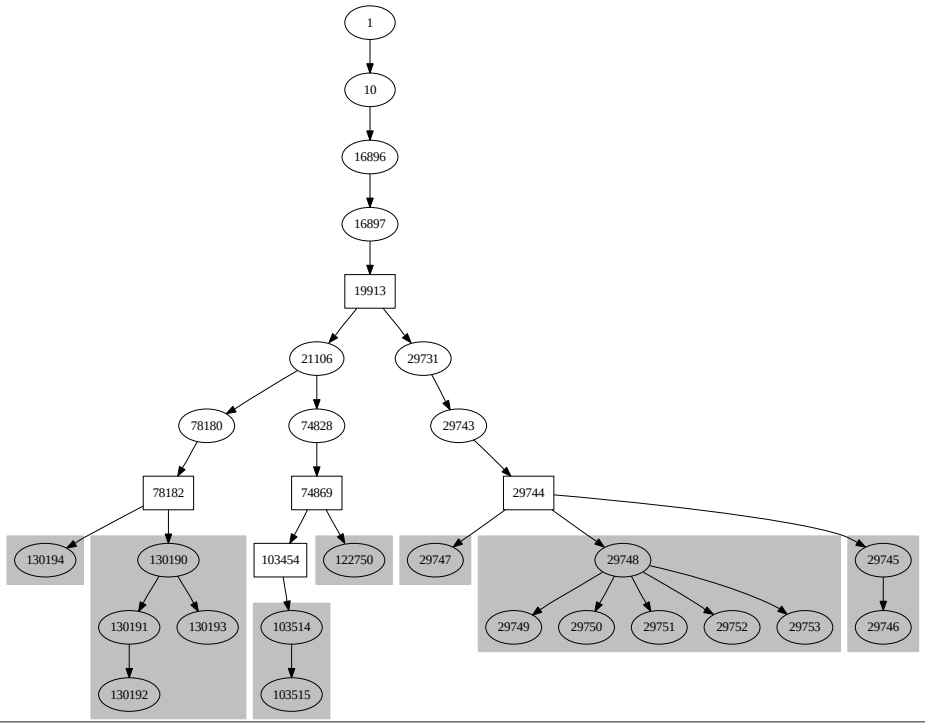


Figure 6. Visualization of topological differences between two Findbugs revisions. The grayed nodes are a subset of those nodes removed by Stage 2. Rectangular nodes represent modified methods. Our case study investigates the cause behind the differences for the subtrees rooted at nodes 29747, 29748, and 29745.

App. Name	# nodes old rev.	# nodes new rev.	Description
checkstyle	1127149	1127299	a tool to help programmers write Java code that adheres to a coding standard
doctorj	360270	360276	javadoc analysis tool
findbugs	155787	153518	uses static analysis to look for bugs in Java code
jaranalyzer	569	548	dependency management utility for jar files
java2html	3177	1366	Java code to html convertor
jruby	113796	125436	Java implementation of the Ruby programming language
jython	69648	70741	Java implementation of the Python programming language
pmd	481033	481034	scans Java code and looks for potential problems (bugs, dead code, ... etc.)

Table 1. Applications evaluated with their CCT size for two revisions.

Finally, we investigate the reason for the total execution time difference between the two CCTs (with topological differences excised). We find that the node with the highest difference in $pweight$ is of the method *PreorderVisitor.visitCode()*. PARCS reports that both the execution time and invocation count has changed in the new revision. The invocation count drops from 8469 in the old revision to 5427 in the new one. PARCS reports that this method invokes a call to *OpCodeStackDetector.visitCode()* that was removed in the new revision and added to the caller of *PreorderVisitor.visitCode()*. This change causes a drop in the invocation count of that method which decreases its total execution time.

App. Name	# old files	# new files	# old methods	# new methods	deleted methods	added methods	modified methods	renamed methods
checkstyle	1386	1386	11948	11953	3	8	11	0
doctorj	226	226	3934	3937	2	5	4	0
findbugs	3570	3569	27424	27415	12	3	11	0
jaranalyzer	413	423	3397	3486	26	115	587	0
java2html	121	132	819	873	138	192	302	0
jruby	4156	4259	25514	26653	773	1912	1592	0
jython	1819	1820	15487	15520	0	33	39	0
pmd	923	923	11336	11336	4	4	6	0

Table 2. Parameters and results of bytecode comparison.

5. Experimental Evaluation

Our experimental platform is a dual-core Intel Core 2 Duo machine clocked at 2.4 GHz with 4M of L2 cache and 2GB of main memory running Linux-2.6.24. The Java virtual machine used is HotSpot version 13.0-b02 within OpenJDK 1.7.0. We extended our JVM to instrument and collect performance statistics and method calls and returns. We ignore calls to the JVM runtime and to the Java libraries for efficiency.

Table 1 describes the eight open-source Java applications that we use to evaluate PARCS empirically. For each application, we use PARCS to compare the dynamic behavior of two close yet stable revisions of the code running with the same test input. For some applications we compare releases instead of revisions because revisions are unavailable (jaranalyzer, java2html and jruby). We used Apache Byte Code Engineering Library (BCEL) [8] to perform bytecode comparison of revisions. The second and third column of the table show the CCT size, in number of nodes, of the old and new revisions, respectively, for each application.

The test input that PARCS employs for CCT creation determines the amount of topological and weight difference found. Software developers can expose more differences by orchestrating special test inputs. For our experiments, we construct the input ourselves or employ one provided by the application repository. In addition, we plan to investigate using multiple inputs (multiple CCT comparisons) and employing automatic test input generation as part of future work. We plan to make all of our inputs, JVM modifications, and BCEL tool available should this paper be accepted.

5.1 Bytecode Comparison

We first quantify the changes that PARCS finds when performing method-level bytecode comparison between two revisions using Table 2. Columns two and three show the number of unique class files from each application code base. Columns four and five show the number of methods in the old and new revisions, respectively. Columns six to nine contain the difference in terms of methods added, deleted, modified and renamed. The highest numbers belong to jaranalyzer, java2html and jruby, for which we use releases instead of revisions.

PARCS finds no renamed methods for any of the applications. This is because of the strict definition of a renamed method that we adopt in which only the method name should change. During our tests, we have found that a method name change is always accompanied by a change in the signature or the containing class, which we classify as a method removal then addition (Section 3.2).

5.2 Topological Difference

To evaluate the common-tree matching algorithm that PARCS employs, we quantify the total number of subtrees and nodes that PARCS removes from both trees at each stage. Relaxed common-tree matching proceeds by comparing nodes of the two trees level-by-level. The algorithm does not proceed to compare subtrees rooted at unmatched nodes, and reports them as a topological difference. After all topological differences are removed, what remains is the common-tree which is the intersection of the two CCTs compared. We have quantified the size of the common-tree obtained for each application. The bigger the common-tree size, the more topologically similar the CCTs are.

We show the results in Table 3. The third columns is the common-tree size as percentage of the CCT size of the old revision. Six of the eight applications show high similarity (above 89%). Pmd shows the highest similarity as only one node is reported as a topological difference. As we mention previously, we compare releases for jaranalyzer, java2html and jruby. As expected, java2html and jruby show low similarity, while jaranalyzer shows high similarity between its releases. These empirical results indicate that relaxed common-tree matching is effective for comparing software close revisions and, though being conservative, does not delete large portions of the trees as the case for releases and distant revisions.

The other columns show the number of subtrees and the equivalent number of nodes that PARCS removes at each stage. The columns titled “added” and “deleted” contain data about subtrees removed due to being rooted at added or deleted nodes (Section 3.2). The one titled “modified” contains trees that have at least one modified node as a dominant node. “Side effects” are unmatched subtrees that cannot be classified as any of the above. Zero values under these columns indicate that no added, deleted or modified methods were executed during execution. Pmd has zeros in all three categories because of the low number of changed methods (Table 2).

We also studied the benefit of using CCTs with call-site information. As mentioned before, call-site CCTs contain more information as they distinguish methods called from the same context but from different call-sites. This extra information becomes useful for topological comparison as it reveals more differences that would have been, using ordinary CCTs, aggregated with other subtrees.

App. Name	common tree size	common tree size (%)	deleted		added		modified		side effects	
			subtrees	nodes	subtrees	nodes	subtrees	nodes	subtrees	nodes
checkstyle	1122368	99.58	159	1059	80	521	984	4743	522	3389
doctorj	360249	99.99	0	0	6	18	7	30	0	0
findbugs	135528	87.00	19	19856	5	17967	29	418	2	8
jaranalyzer	538	94.39	0	0	8	10	8	10	8	22
java2html	421	13.25	42	1314	37	608	23	1779	0	0
jruby	8804	7.74	1811	97983	2159	93883	1467	29768	0	0
jython	62955	89.29	0	0	786	6671	1395	5600	276	3067
pmd	481033	100.00	0	0	0	0	0	0	1	1

Table 3. Subtrees and nodes removed at each stage of topological differencing.

App. Name	deleted	added	modified	side effects
checkstyle	76	66	102	0
doctorj	0	4	6	0
findbugs	19	5	28	2
jaranalyzer	0	4	4	5
java2html	25	20	20	0
jruby	1196	1754	808	0
jython	0	125	218	37
pmd	0	0	0	1

Table 4. Subtrees removed at each stage for CCTs without call-site information.

To assess the additional differences revealed via call-site CCTs, we have compared the number of subtrees removed as topological differences using both types of CCTs. Higher number of subtrees removed means more differences that PARCS discovers. Table 4 summarizes our results. Similar to Table 3, it shows the numbers of subtrees removed divided into the four classifications. For most applications, the difference is significant. For example, checkstyle has 522 subtrees removed as side effects using call-site CCT Table 3. This number drops to zero when switching to a conventional CCT. This indicates that all those subtrees are merged under other subtrees (of the same context and root method name) that were successfully matched, and thus the difference is hidden. This case occurs when the same method is invoked multiple times from the same context from different call sites. This effect is not present for FindBugs and hence the numbers are nearly identical.

The trade off that we make for this increase in detail (and thus understanding of program behavior) is in the CCT size. In Table 5, we quantify this overhead for both revisions of our applications. Columns two and three show the CCT size as the number of nodes, with and without call-site information. The fourth column is the percent increase in CCT size due to using call-site information. The old revision of java2html shows the highest increase (478.69%) while jaranalyzer's new revision shows the lowest (21.24%). The average increase is slightly more than the original size of the CCT. The data show that the increase is nearly similar across revisions. The only two exceptions are java2html and jruby, for which we employ releases instead of revisions.

App. Name	# nodes w call-site	# nodes w/o call-site	Difference (%)
checkstyle_old	1127149	307424	266.64
checkstyle_new	1127299	307544	266.55
doctorj_old	360270	260061	38.53
doctorj_new	360276	260064	38.53
findbugs_old	155787	98979	57.39
findbugs_new	153518	97222	57.90
jaranalyzer_old	570	465	22.58
jaranalyzer_new	548	452	21.24
java2html_old	3177	549	478.69
java2html_new	1366	469	191.26
jruby_old	113803	68694	65.67
jruby_new	125439	85545	46.64
jython_old	70507	33840	108.35
jython_new	70741	33494	111.20
pmd_old	481033	375506	28.10
pmd_new	481034	375507	28.10
			Avg = 114.21%

Table 5. Comparison of CCT size with and without call-site information.

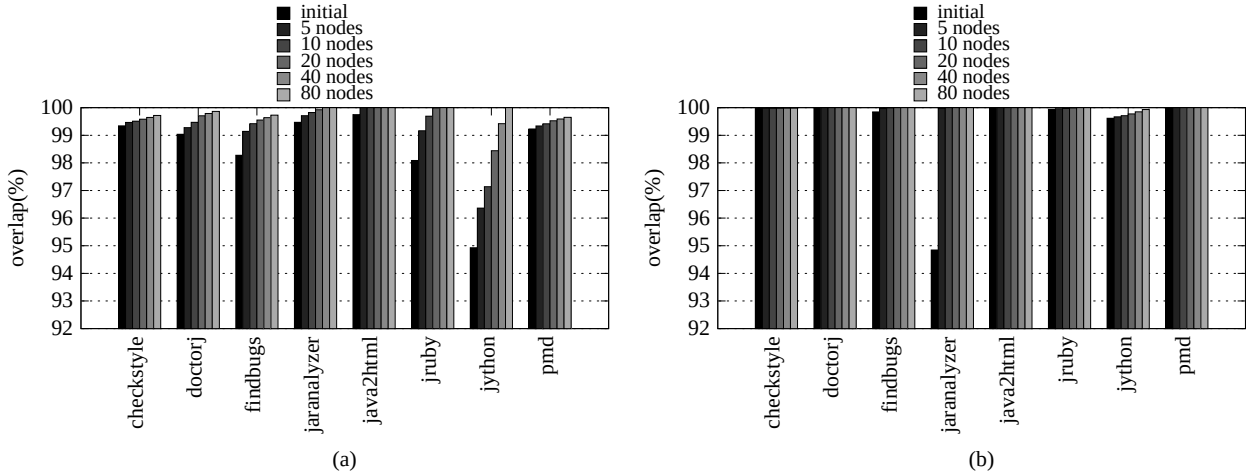


Figure 7. Initial and weight-adjusted Overlap for metrics execution time (a) and invocation count (b).

5.3 Weight Difference

After PARCS removes all topological differences from both CCTs, the remaining trees are topologically identical. This means that if the two trees are traversed in some order (e.g. breadth-first), the nodes encountered at the k^{th} step in both trees will be identical in terms of method name and context. Other parameters, however, may vary, such as call-site, average execution time and invocation count.

In this section, we quantify the weight similarity of the two identical CCTs for each application using the overlap metric from in Section 3.4. We first compute the initial overlap of the two trees, then we study the effect of weight adjustment. We consider two types of weights in this evaluation: invocation count and total execution time. The invocation count is how many times a method (node) was called from specific call-site. The total execution time is the average execution time multiplied by invocation count.

Figure 7 presents these results. The left graph is when we use execution time as the weight and the right graph is when we use invocation count as the weight. The graphs show the initial overlap and how it increases after we weight-adjust the top 5, 10, 20, 40 and 80 node pairs. The increase in overlap is highly monotonic for execution time. Also, the initial overlap is high (above 94%). This indicates that the node matching found by the relaxed common-tree matching is very accurate (and that the pair are likely to be semantically similar), since they exhibit similar execution time. The initial overlap for invocation count is nearly 100% for most applications. This is expected since invocation counts should vary more on change of user inputs rather than on functional upgrades. However it is still a useful and interesting metric to consider.

6. Related Work

Since the CCT data structure was originally proposed [2], much research has been contributed that decrease its size and collection overhead. In [14], the idea of the Partial Calling-Context Tree (PCCT) is proposed. A PCCT is partial because the tree is built using sampling of the runtime stack to a certain depth and updating the tree accordingly. The idea was further used in [5, 6, 4, 17] and extended with more efficient sampling schemes that decrease overhead while maintaining high accuracy of the approximate CCT. The accuracy metric used, however, is the overlap (Section 3.4) of the approximate CCT with the full CCT, which does not capture the significance of the topological differences between the two CCTs. In other words, if a number of short-running methods is missing from the approximate CCT, the accuracy will be slightly affected while the change in performance could be drastic. Also, call-site information is ignored for sake of size reduction. Since our approach is entirely offline, we chose to build a full CCT with call-site information incorporated.

In [3, 11, 12, 13], algorithms for syntactical, semantic, and structural comparison of software versions is proposed. All of these prior works, however, operate statically. This is different from our approach, since we rely on dynamic profile (CCT) generated by test runs of the application. Relying on dynamic profile can expose unforeseen effects of code modifications that are hard to identify using only static analysis. Our approach thus complements these efforts.

Zhang et al. propose a technique to match entire execution histories of two program versions running with the same input [15]. The execution history contains control flow taken, values produced, addresses referenced and data dependences. This is different from our technique since these prior works assume semantically equivalent versions (e.g. optimized and unoptimized) while we compare different revisions of a program that can include functional upgrades.

The work most similar to ours is described by Zhuang et al. in [16]. They have developed a framework for comparing CCTs of the same program when running on different platforms (compilers, runtimes, systems) or with different inputs. They rely on the tree transformation algorithm proposed in [1] to perform the comparison efficiently. While this approach is useful to quantify the difference in execution on different platforms or when using different inputs, it is not suitable for comparing functionally different versions of the program. Furthermore, due to the nature of the tree transformation technique they adopt, the nodes matched from both trees are not necessarily semantically equivalent. We have discussed this limitation further in Section 2.2.1. Our work is the first, to our knowledge, to focus on revision-based dynamic behavior and performance differences with support of source code repository systems.

7. Conclusion and Future Work

In this paper, we present PARCS, an offline analysis tool that automatically identifies differences between the execution behavior of two revisions of an application. PARCS collects program behavior and performance characteristics via profiling and generation of calling context trees (CCTs). We annotate CCTs with call-site information and performance metrics to facilitate identification of differences in CCT topology (changes in the calling patterns of the program) and in overall program performance (via weight differencing). We overview our techniques for identifying differences in CCTs and demonstrate how we use PARCs to attribute differences in execution behavior and performance to specific changes in the source code.

We have presented an empirical evaluation of PARCS using a number of well-known Java applications. We present what supports the use of call-site information to expose additional topological differences than conventional CCTs. We also quantify topological and weight differences between two revisions of each application. Our results show high topological similarity between close revisions with changes constituting less than 15% of the CCT size. For applications for which only releases were available to us (releases constitute a very large number of revisions thus, many changes), we find much less similarity. This result emphasizes the importance of using PARCS for small, minor changes to track and gain a better understanding of how software updates impact overall behavior and performance. Moreover, our approach to weight matching to identify differences in performance metrics (node weights) has greater than 94% overlap for all applications indicating that relaxed common-tree matching works well for revision comparison. Overall, we find that PARCS is most effective for incremental changes such as those common to revisions. As such PARCS has the potential for facilitating improved understanding of the behavior and performance of complex software systems and their evolution over time.

As part of future work, we are investigating ways to generate inputs automatically and whether employing multiple inputs (CCT comparisons) is effective to target and attribute behavioral differences. In addition, we are considering coupling PARCS with static analysis to reveal more semantic information about the program and code changes. Finally, we are working on automating the identification of modification differences (Stage 2). With such support, only differences due to side-effects and non-determinism will have to be investigated by hand. The PARCS framework and visualization of behavior and performance data however, significantly simplifies this process.

References

[1] ALBERTO APOSTOLICO, Z. G. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[2] AMMONS, G., BALL, T., AND LARUS, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation* (New York, NY, USA, 1997), ACM, pp. 85–96.

[3] APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 2–13.

[4] ARNOLD, M., AND GROVE, D. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *CGO '05: Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 51–62.

[5] ARNOLD, M., AND RYDER, B. G. A framework for reducing the cost of instrumented code. *SIGPLAN Not. 36*, 5 (2001), 168–179.

[6] ARNOLD, M., AND SWEENEY, P. F. Approximating the calling context tree via sampling. Tech. rep., IBM Research, July 2000.

[7] BOND, M. D., AND MCKINLEY, K. S. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications* (New York, NY, USA, 2007), ACM, pp. 97–112.

[8] Bytecode engineering library. `http://jakarta.apache.org/bcel/`.

[9] FELLER, P. T. Value profiling for instructions and memory locations. Master's thesis, University of California, San Diego, April 1998.

[10] Findbugs. `http://findbugs.sourceforge.net/`.

[11] JACKSON, D., AND LADD, D. A. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* (Victoria, BC, Canada, 1994), IEEE Press.

[12] LASKI, W., AND SZERMER, J. Identification of program modifications and its applications in software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance* (Victoria, BC, Canada, 1992), IEEE Press, pp. 282–290.

[13] MYERS, E. W. An o(nd) difference algorithm and its variations. *Algorithmica 1* (1986), 251–266.

[14] WHALEY, J. A portable sampling-based profiler for java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande* (New York, NY, USA, 2000), ACM, pp. 78–87.

[15] ZHANG, X., AND GUPTA, R. Matching execution histories of program versions. *SIGSOFT Softw. Eng. Notes 30*, 5 (2005), 197–206.

[16] ZHUANG, X., KIM, S., IO SERRANO, M., AND CHOI, J.-D. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2008), ACM, pp. 4–13.

[17] ZHUANG, X., SERRANO, M. J., CAIN, H. W., AND CHOI, J.-D. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not. 41*, 6 (2006), 263–271.