Tracking Performance Across Software Revisions

Nagy Mostafa Chandra Krintz
Computer Science Department
University of California, Santa Barbara
{nagy,ckrintz}@cs.ucsb.edu

ABSTRACT

Repository-based revision 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 for "performance-aware" repository and revision control for Java programs. Our system 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 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.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; C.4 [**Performance of Systems**]: *Performance attributes*

Keywords

performance-aware revision control, profiling, calling context tree

1. INTRODUCTION

Software developers world-wide employ revision control (RC) systems for managing a vast 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 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. Some RC systems provide additional 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, 16, 6, 4, 19] and performance differencing [18], but is unique in that it targets two different revisions 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 [18]. 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 compares the trees using common tree matching that we extend with feedback from changes that developers have 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 categories that distinguish a likely reason behind the performance differences: method addition/deletion, di-

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.

rect 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 topologically identical. PARCS analyzes these trees for differences in the performance metrics that annotate their nodes. 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 each topological and weight difference to a set of probable 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 behavioral differences between two revisions of these applications. 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 Java programs.

We next provide background on the techniques that PARCS builds upon and extends. We then detail the PARCS design and implementation (Section 3) and present a case study on our use of PARCS for revisions of the popular FindBugs application (Section 4). 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 in 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 Performance Profiling and Representation

PARCS collects, manipulates, and compares the dynamic behavior of a program using a data structure called calling context tree (CCT) [2]. Given a method call stack, the calling context is the list of methods that are resident on the stack at any particular time. A CCT captures the calling context of each dynamic method invocation that occurs during program execution. All activations of the same method that execute from the same calling context are aggregated into a single node. An edge X→Y represents a call from method X to method Y. The calling context of node Y, thus, is captured by the series of nodes from the root of the tree down-to node Y. A CCT edge can be annotated with various execution metrics of the call it represents, such as invocation count, average execution time, ..., etc.

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

```
public void A(){
  1: B();
     ...
  5: C();
     ...
  9: B();
}
public void B(){
  1: D();
     ...
  5: E();
}
public void C(){
  1: F();
}
public void D(){}
public void E(){}
public void F(){}
```

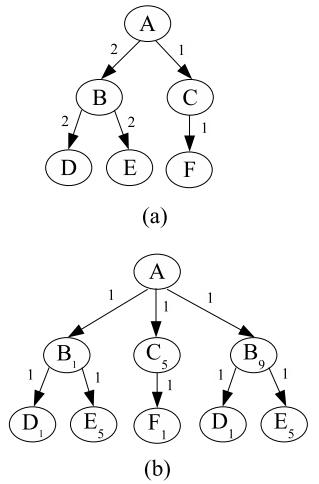

Figure 1: Code snippet with the corresponding CCT. (a) The corresponding CCT with no call-site information included. (b) The equivalent tree with call-site information shown as subscripts

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 the same activation of method X to have the same context [18]). 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. Call-site information can also be used for measuring code coverage and anomaly detection [8]. 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 points of failure. Moreover, CCTs provide a good trade-off between size and accuracy compared to dynamic call trees (DCTs) and dynamic call graphs (DCGs) [2].

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 total and average execution time and invocation count. The PARCS framework is extensible enabling researchers to investigate its efficacy using other performance metrics (e.g. cache misses, branch mispredictions).

2.2 Identifying Topological Differences

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 trans-

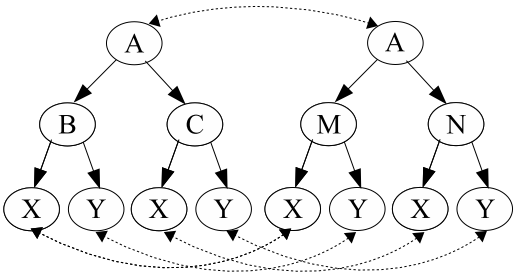

Figure 2: Example of tree transformation. Two rename operations needed to transform the left tree to the right one.

formation 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 to one tree, transforms it to the other. The algorithm is proven optimal in the number of transformation operations used (the edit distance) and has a time complexity of $O(|CCT1| \times |CCT2|)$. The transformation 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 it to compare two CCTs for the *same* program that they execute on different platforms or with different inputs [18]. 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 minimum sequence of operations that transforms the left CCT to the right one. 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.

There are two drawbacks, however, that discourage us from adopting this algorithm for PARCS. First, 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, unless C was renamed to N as part of code modification, method X called by C has a different semantic than X called by N. Considering the two to be equivalent could be misleading to performance analysts. Since in PARCS we are comparing two revisions of a program, these inaccuracies are more likely to occur more often since the code is different across revisions. Inaccuracies in PARCS lead to incorrect identification of differences and attribution of differences to code changes. Second,

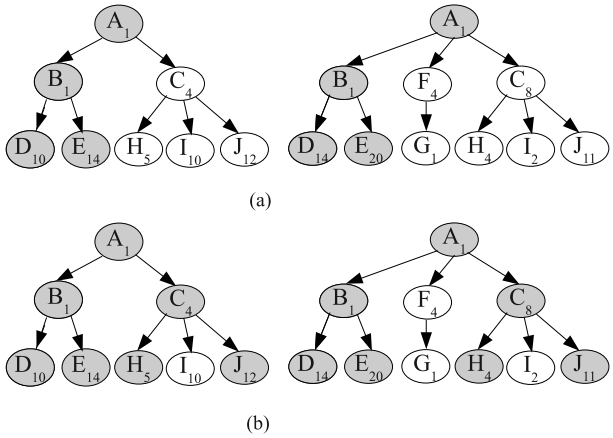

Figure 3: Common tree matching examples. (a) Strict common tree matching. (b) Relaxed common tree matching. Subscripts are call-sites. Shaded nodes form the common tree found in each case

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 185,960 nodes on average. Thus a matrix of over than 34 billion entries is needed. Empirically, the algorithm runs out of memory for 80% of our test cases. For the above reasons, we investigate an alternative approach to CCT matching that is more semantically aware and suitable for large CCTs.

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 n ($n.order$) 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 ($\equiv$):
Given $n_1 \in CCT_1$ and $n_2 \in CCT_2$, $n_1 \equiv n_2$ iff

1. $n_1.method\_name = n_2.method\_name$ and

2. $n_1.order = n_2.order$ and

3. $n_1.parent \equiv n_2.parent$

This definition implies that equivalent nodes always have the same context. Moreover, if a node has no equivalence, we consider the subtree rooted at it a topological difference.

Figure 3 (a) illustrates a common tree matching example (subscripts are call-sites). 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. We proceed similarly for the last level. The shaded 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 seems to have been modified to call method F before it calls C shifting C one step to the right. 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 definition above to use the relative ordering among matched nodes instead. Our definition of equivalence then becomes:

Definition 2. Relaxed Node Equivalence($\equiv_R$):
Given $n_1, p_1 \in CCT_1$ and $n_2, p_2 \in CCT_2$.
Let L_1 and L_2 be the set of left siblings of n_1 and n_2, respectively.
Let (p_1,p_2) be the last pair of equivalent nodes found (if any).
Then $n_1 \equiv_R n_2$ iff

1. $n_1.method\_name = n_2.method\_name$ and
2. $n_1.parent \equiv_R n_2.parent$ and
3. $p_1 \in L_1 \Leftrightarrow p_2 \in L_2$ and
4. $p_1.call\_site < n_1.call\_site \Leftrightarrow p_2.call\_site < n_2.call\_site$

We refer to the version of the algorithm that employs this definition of equivalence as *relaxed common-tree matching*. Using this algorithm, equivalent nodes still have the same context (Rule 2). The difference is that they do not have the exact child order. This is relaxed by Rules 3 and 4. Rule 3 means that for each pair of equivalent nodes, a common subsequence of nodes is found from the two sequences representing their ordered children. This takes care of cases where extra calls shift nodes among their siblings. Rule 4 adds the constraint that the relationship ($< or >$) between two consecutive children call-sites in the two sequences must be identical (note that no two different nodes can have the same parent and equal $call\_site$). This ensures that actual shifting of nodes has taken place and that we are not matching to a wrong node that happens to have the same method name.

Figure 3 (b) illustrates how the relaxed common tree matching works. Despite having the same sibling order, nodes I_{10} and I_2 are not matched merely because the call site of I_{10} is greater than that of H_5 which is not the case for I_2 and H_4. This means that although the subtree at C has been shifted due to code modification (the call to F), the two invocations of I are different. On the other hand, node C is matched despite of the different sibling order and the subtree rooted at F is reported as a difference. We employ relaxed common-tree matching within PARCS to identify topological differences between two CCTs (program revisions). We will quantify in Section 5 the improvement in matching accuracy that relaxed common tree matching achieves over conventional common tree matching.

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.

In addition, PARCS performs a fast, static bytecode comparison on the revisions to extract method-level changes. PARCS feeds the CCTs and the 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 PARCS to include library calls if necessary. 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

We perform 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 a 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 its 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.

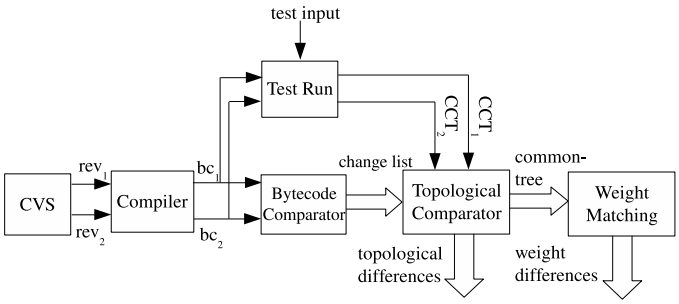

Figure 4: PARCS process. rev_1 and rev_2 refer to old and new revisions, respectively. bc_1 and bc_2 are the corresponding bytecodes.

- **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:* Any randomness in 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:

Stage 1. We excise all subtrees rooted at added and deleted nodes 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. 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 by method signatures and contexts. We ignore unmatched dominators for now 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 and excise the subtrees afterwards as they become identical. We report all differences found as potentially resulting from direct modification since they are dominated by at least one modified node. 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.

Stage 3. Finally, we conduct a global topological comparison for what remains of the two CCTs and excise unmatched subtrees. These subtrees are present either due to side effects or non-determinism (Reasons 3 and 4) as they are not dominated by any modified nodes. In other words, none of their callers is a modified method, so the reason they are present in one revision and not the other is either they were enabled by an indirect effect of code modification or due to randomness of execution. We excise (and report) all unmatched subtrees.

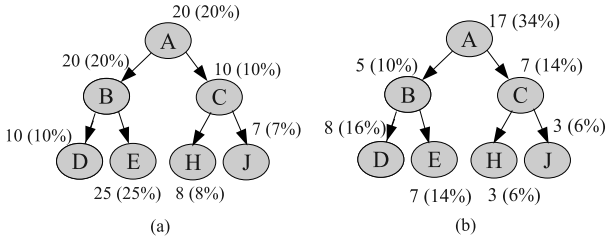

Figure 5: Weight matching example. Common trees with identical topology and different weights.

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 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, 18]. We define overlap in our setting as:

$$overlap(CCT_1, CCT_2) = \sum_{\substack{n_i \in CCT_1 \\ n_j \in CCT_2 \\ n_i \equiv_R n_j}} min(pweight(n_i, CCT_1), pweight(n_j, CCT_2))$$

where $n_i \equiv_R n_j$ means that n_i in CCT_1 is equivalent, under relaxed equivalence definition, to n_j in CCT_2 (the two nodes match). We define $pweight(n, CCT)$ as the percentage that the weight of node n constitutes out of the total weight of all nodes in CCT. 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 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, has a 99.3% overlap in execution time between two identical runs. 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 [18]. The algorithm performs weight adjustments to improve the overlap up to a pre-defined threshold, the nodes adjusted are the ones with most significant weight difference. The algorithm is parameterized by an overlap threshold and/or number of nodes of interest. 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. The nodes that PARCS returns are the methods responsible for the greatest contribution to the overall weight difference between the two revisions.

3.5 Attributing Differences

In our current prototype of PARCS, we report each difference with an ordered list of methods that most likely contain the code change(s) that caused 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 exact attribution of a difference to a specific change 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. 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 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 draw the modified methods as rectangular nodes. The nodes in gray are those that Stage 2 identifies as a difference 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 excised subtrees. PARCS orders the list from the modified node nearest to the subtree to the farthest. For the subtree rooted at node 29748, 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 described earlier, 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 induces 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 the method's caller, 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 when the class initializer of *JavaVersion* is invoked to cause topological difference.

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 have 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()* instead. This change causes a drop in the invocation count of that method which decreases its total execution time.

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 with our extensions for collection of performance statistics and context profiles.

App. Name	Description
checkstyle	Code style checker for Java. SVN revisions 2090 and 2100
doctorj	Javadoc analysis tool. Versions 5.1.1 and 5.1.2
findbugs	Bug detector for java. CVS commits on 21st and 25th Aug. 2008
jaranalyzer	Jar files dependency analyzer. Versions 1.1. and 1.2
java2html	Java to Html convertor. Versions 4.1 and 4.2
jruby	Java implementation of Ruby. Versions 1.1.2 and 1.1.3
jython	Java implementation of Python. SVN revisions 4899 and 4981
pmd	Java code checker. SVN revisions 6399 and 6421

Table 1: Description of applications studied.

Table 1 describes the eight open-source Java applications that we use to evaluate PARCS empirically along with the revisions/versions used. For each application, we use PARCS to compare the dynamic behavior of two close revisions of the code running with the

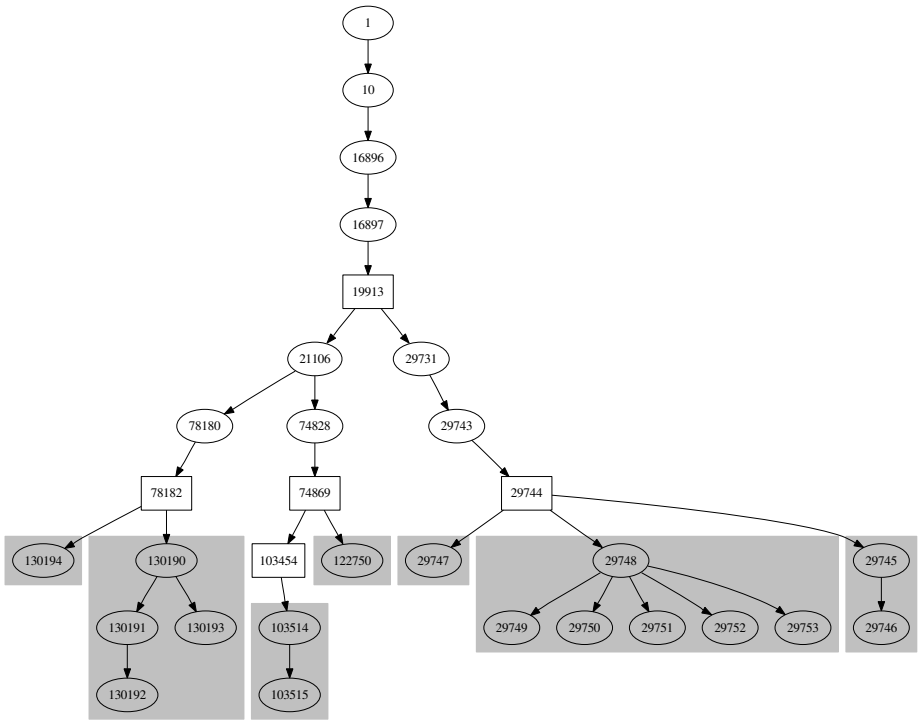

Figure 6: Visualization of topological differences between two Findbugs revisions. The shaded substrees are a subset of those subtrees removed by Stage 2. Rectangular nodes represent modified methods.

App. Name	Average Node Count		Average Depth	
	old	new	old	new
checkstyle	1448937.6	1449183.6	31.76	31.81
doctorj	390814.2	390816.4	30.87	29.94
findbugs	185960.2	183552.2	19.85	19.74
jaranalyzer	569.6	565	16.58	16.28
java2html	2886.6	1075.4	10.73	10.29
jruby	321365.2	304357.6	34.54	31.95
jython	145700.4	145896.2	21.3	21.26
pmd	676247.4	676248.4	53.5	53.28

Table 2: Applications average CCT sizes and average stack depth for two revisions.

same test input. Revisions were chosen to be no more than ten days apart. For four applications (doctorj, jaranalyzer, java2html and jruby) we compare releases instead of revisions because we did not have access to the revision control system. We used doctorj, findbugs, jaranalyzer, java2html and PARCS itself as inputs for checkstyle, java2html and pmd. For jruby and jython, we used five microbenchmarks: binarytrees, nsievebits, fannkuch, mandelbrot and nsieve from The Computer Language Benchmarks Game [14]. For findbugs, we used checkstyle, java2html, PARCS and two other java projects. And finally for jaranalyzer, we used checkstyle, findbugs, jruby, jython and an orchestrated test case.

Table 2 shows the average CCT size, in number of nodes, and time-weighted average stack depth of the old and new revisions for each application over five inputs. The numbers are close indicating the high similarity of the revisions.

5.1 Bytecode Comparison

We used Apache Byte Code Engineering Library (BCEL) [7] to perform method-level bytecode comparison of revisions. We quantify the results the comparison in Table 3. Columns two and three show the number of 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 deleted, added, 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).

App. Name	OF	NF	OM	NM	DM	AM	MM	RM
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 3: Parameters and results of bytecode comparison. OF:old files, NF:new files, OM:old methods, NM:new methods, DM:deleted methods, AM:added methods, MM:modified methods, RM:renamed methods

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. We also measure the size of the common-tree obtained for each application for each input.

We show the results in Table 4. 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 common tree coverage (above 85%). Pmd shows the highest common tree ratio as only one node

App. Name	common tree size	common tree size (%)	deleted		added		modified		side effects	
			subtrees	nodes	subtrees	nodes	subtrees	nodes	subtrees	nodes
checkstyle	1105139	99.9	157	994	159	1217	31	83	18	148
	1633395	99.86	248	2172	266	2639	59	157	2	2
	1856117	99.9	308	1615	318	2014	56	139	16	104
	630763	99.97	137	137	137	137	0	0	16	68
	2009773	99.8	473	3745	509	4483	86	222	18	156
doctorj	393342	99.99	1	13	7	20	5	15	0	0
	609205	100	0	0	2	2	4	14	0	0
	108601	99.99	0	0	3	3	6	21	0	0
	207495	100	0	0	0	0	0	0	0	0
	635365	100	1	14	6	20	5	15	0	0
findbugs	135149	86.96	19	19856	6	17970	38	436	0	0
	129333	87.55	18	18079	6	16235	38	350	0	0
	187604	86.86	20	27824	6	25779	40	587	0	0
	189830	87.88	20	25610	6	23565	38	596	1	1
	168807	86.71	20	25039	6	23339	36	850	0	0
jaranalyzer	507	97.31	0	0	11	14	11	14	0	0
	567	97.09	0	0	11	14	11	14	2	6
	558	97.55	0	0	11	14	11	14	0	0
	586	97.5	0	0	11	14	12	15	0	0
	534	93.68	0	0	11	14	11	14	8	22
java2html	216	7.48	9	2671	11	853	3	7	0	0
	218	7.55	9	2671	11	853	3	7	0	0
	224	7.74	9	2671	11	853	3	7	0	0
	216	7.48	9	2672	11	853	3	7	0	0
	203	7.06	9	2671	11	853	3	7	0	0
jruby	100385	7.21	37189	884560	81821	941162	69716	661326	0	0
	9884	18.75	282	34619	578	42306	1752	10816	0	0
	9221	17.08	220	41268	459	46603	1706	4823	0	0
	9616	18.92	227	35403	453	41565	1794	7380	0	0
	9982	17.39	223	42545	486	49768	1779	6294	0	0
jython	486326	98.54	0	0	760	6830	1512	7615	10	192
	40951	85.08	0	0	755	6799	1510	7557	6	163
	54603	88.38	0	0	756	6799	1512	7613	6	163
	53554	88.08	0	0	764	6866	1511	7562	14	224
	56948	88.63	0	0	771	6923	1522	7673	16	240
pmd	526553	100	0	0	1	1	0	0	0	0
	931798	100	0	0	1	1	0	0	0	0
	826271	100	0	0	1	1	0	0	0	0
	817248	100	0	0	1	1	0	0	0	0
	279367	100	0	0	1	1	0	0	0	0

Table 4: Subtrees and nodes removed at each stage of topological differencing.

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 common tree coverage, while jaranalyzer shows high coverage between its releases.

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.

Figure 7 shows the amount of overlap obtained by PARCS for each test case using two overlap metrics: average execution time (a) and invocation count (b). The results are all high for both (above 82%). As expected, the overlap is less using average execution time due to noise in measurement. High overlap, however, does not mean a good match. For example, java2html shows nearly perfect overlap for both metrics yet, as Table 4 shows, it yields around 7% common tree. Hence, overlap alone can be misleading as it tends to be high if the common tree is small.

To better measure the matching accuracy, we define the following matching score metric:

$$score(CCT_1, CCT_2) =$$
$$(\prod_{o_i \in O} o_i(CCT_1, CCT_2)) \times \frac{|CT|}{|CCT_1|} \times \frac{|CT|}{|CCT_2|} \times 100$$

where $|CT|$ is the common tree size and O is the set of overlap metrics used. In our case, we use three overlap metrics: invocation count, average execution time, and total execution time. Informally, a match with high score is a match whose common tree covers high percentage of the CCT and has high overlaps as well. Figure 8 shows the scores for all applications and all inputs. 6 out of 8 apps exhibit very high score (above 70%). Since we are comparing versions for jruby and java2html, they show very poor matching scores (below 4%). This demonstrates that PARCS is highly tailored for close revisions comparison where code modifications are incremental. Nevertheless, for jaranalyzer the score is surprisingly high even though we are comparing releases.

We next compare matching scores of relaxed common tree matching against strict matching. Figure 9 shows the average scores over all inputs for each application. Relaxed common tree matching shows significant improvement over strict matching for jaranalyzer, jython, findbugs and jruby. For the remaining cases, there is still slight improvement.

We also studied the benefit of using CCTs with call-site information. To assess the additional differences revealed via call-site CCTs, we have compared the number of nodes removed as topological differences using both types of CCTs. Higher number of nodes removed means more differences that PARCS discovers. Table 5 shows the average number of nodes removed for each application over five inputs. For most applications, the difference is significant. For example, checkstyle has 2404 nodes removed which gets nearly doubled when using call-site CCT. jruby shows drastic increase in

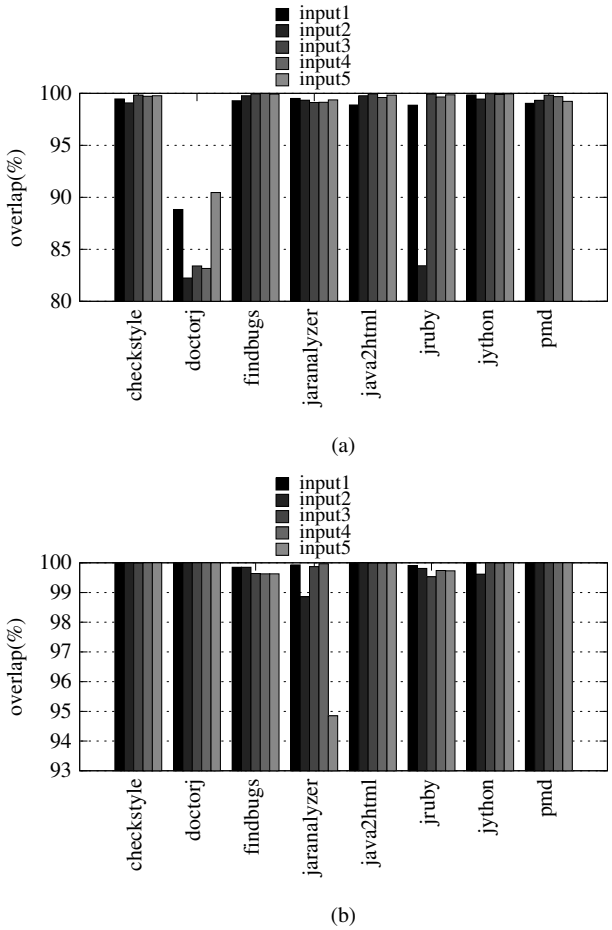

Figure 7: Overlap for two metrics: (a) Average execution time. (b) Invocation count.

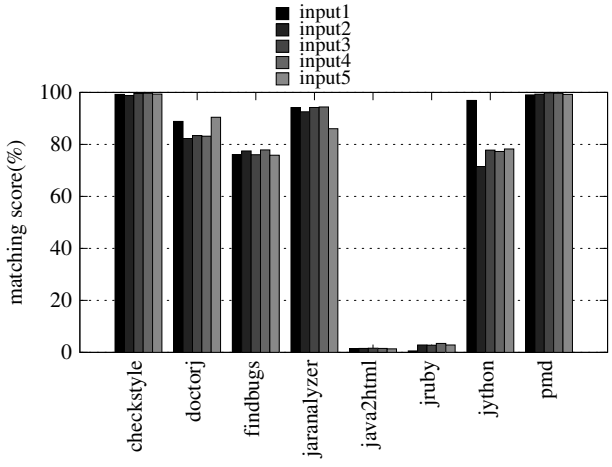

Figure 8: Matching scores

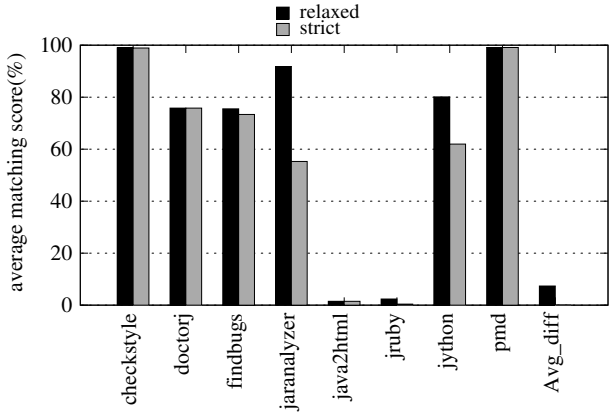

Figure 9: Comparison of matching scores between relaxed and strict common tree matching

App. Name	nodes removed w call-site	nodes removed w/o call-site
checkstyle	4046.4	2404.4
doctorj	27.4	26.4
findbugs	45223.2	23007.8
jaranalyzer	33.8	22.6
java2html	3531.2	440.4
jruby	570087.6	46527.2
jython	14643.8	2621
pmd	1	1

Table 5: Total nodes excised using CCTs with and without call-site information.

node count due to its recursive nature.

The tradeoff that we make for this increase in detail (and thus understanding of program behavior) is in the CCT size. In Table 6, 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 jruby shows the highest increase while jaranalyzer's new revision shows the lowest. The average increase is nearly 300%. From our experience, recursion is the primary reason for CCT size explosion when call-site information is included. To reduce the CCT size, a threshold can be placed on its depth which can be tuned based on the amount of tree comparison detail required.

6. RELATED WORK

In [3, 11, 12, 13], algorithms for syntactical, semantic, and structural comparison of software versions are proposed. All of these approaches, 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 [17]. 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 [18]. They have developed a framework for comparing CCTs of the

App. Name	nodes w call-site	nodes w/o call-site	difference (%)
checkstyle_old	1448937.6	385136.8	276.21
checkstyle_new	1449183.6	385294.0	276.12
doctorj_old	390814.2	273867.6	42.70
doctorj_new	390816.4	273868.8	42.70
findbugs_old	185960.2	120527.6	54.29
findbugs_new	183552.2	118694.6	54.64
jaranalyzer_old	569.6	462.0	23.29
jaranalyzer_new	565.0	459.4	22.99
java2html_old	2886.6	344.8	737.18
java2html_new	1075.4	265.2	305.51
jruby_old	321365.2	25902.6	1140.67
jruby_new	304357.6	32369.4	840.26
jython_old	145700.4	28075.6	418.96
jython_new	145896.2	27695.4	426.79
pmd_old	676247.4	527235.6	28.26
pmd_new	676248.4	527236.6	28.26
			Avg = 294.93%

Table 6: Comparison of CCT sizes with and without call-site information.

same program when running on different platforms or with different inputs. The CCTs they used, however, do not include call-sites which keeps the CCT size reasonable and enables them to use 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 as information gets blurred in the CCT. 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. Moreover, we developed a scoring metric to assess matching accuracy and have shown that PARCS is best applied to revisions comparison to track and gain a better understanding of how software updates impact overall behavior and performance.

As future work, we are investigating the impact of test inputs on performance differences revealed. In addition, we are considering coupling PARCS with program slicing techniques [15] to reveal semantic details about the program to aid in automatic attribution of performance differences to source code changes.

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

[8] 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.

[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] The computer language benchmarks game. `http://shootout.alioth.debian.org`.

[15] WEISER, M. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), IEEE Press, pp. 439–449.

[16] 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.

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

[18] 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.

[19] 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.