Analyzing Performance Differences between Multiple Code Versions

Tomasz Kalbarczyk, Noah Imam, Tianyi Zhang,
Yamini Gotimukul, Brian Boyles, Sushen Patel
The University of Texas at Austin
{tkalbar, n.imam, troyathust, yamini.gotimukul, bboyles, sushenpatel}@utexas.edu

ABSTRACT

Optimizing code performance has always been an important goal of programmers who code at a low-level, and is gaining importance even in high-level programs due to the proliferation of mobile and distributed applications with limited processing and memory resources. Nevertheless, software developers are often ignorant of the way their code modifications affect performance until it is too late or quite expensive to rectify the problem. Diff utilities are a well-established method for reviewing code changes and identifying bugs on enterprise scale software projects, but they only provide information on syntactic changes. We propose a tool, named PerfDiff, that enables developers to quickly identify performance changes at function-level granularity between two versions of a program. Our approach uses an existing profiling tool to obtain performance data on two versions of a program, and then compares the performance of changed functions. The performance measure tracked by PerfDiff is the execution time of the functions.

We have designed the tool in two phases. In the initial phase, to demonstrate the viability of our approach, we have created our own library of C functions with known performance disparities which could be manipulated for testing and evaluation. To establish a proof-of-concept, we have set up a repository from which two versions of code could be automatically pulled to compare their performances using a profiling tool. The implemented tool is successful in providing execution time differences of all interested functions. In the second phase we improved PerfDiff by integrating it with Git commands so that it can run automatically in the background when a user tries to commit the code to the repository. The results of PerfDiff are captured during the commit process to be used for later display. The visualization was also extended so that performance changes can now be viewed by a reviewing developer alongside the syntactic changes of a regular diff. As part of the evaluation of the second phase of our research work, we analyzed the impact of test case run length and background processes on the profiler results and measures that can be taken to mitigate these threats. We also seeded a performance bug explicitly and used PerfDiff to help a developer identify the likely source of the bug. Finally, PerfDiff tool will be improvised to take into consideration the call graph for identification of the performance regression.

Keywords

Performance, Code Differencing

1. INTRODUCTION

Software performance issues are easier to solve the earlier they are discovered in the software development process [1]. The increasing popularity of mobile platforms means that strict performance requirements are not only relevant in the context of mathematically intensive computing, but also in everyday software written in high-level languages. Very often computational resources are under-utilized or used inefficiently, especially in large scale parallel and distributed applications. The factors which determine a program's performance are varied, interdependent, and often hidden from the programmer. Some examples include the algorithms being used, I/O behavior, memory usage patterns, processor architecture, etc. Therefore, providing developers with an intuitive way for quickly and easily diagnosing performance bugs is critical for developing software applications in an efficient manner.

Although interactive differencing tools are now widely used, they do not track the performance changes between versions of software. Developers are often oblivious to such changes since they do not cause any functional differences. This leads to performance bugs permeating through multiple versions of code, and inevitably requiring reversion in order to correct them. If developers had a tool that could immediately highlight the performance impacts of their changes, these issues could be addressed at an early stage of development saving cost, effort and time.To facilitate this capability in PerfDiff, we have designed the tool to post the results as inline comments to Github repository immediately after the code has been pushed to the repository. To show the effectiveness of our tool, as an evaluation strategy, we recorded the performance of the application under test by explicitly seeding an obscure performance bug and tested it twice, once manually and once by running our tool. The results of our evaluation showed that PerfDiff successfully aids a user in narrowing down the source of performance bugs, by identifying faulty functions beyond just showing faulty test cases.

2. RELATED WORK

During the past decade, many researchers and practitioners have contributed to the area of Software Performance Engineering (SPE). In this section, we discuss existing tools and research related to this area.

Performance Regression Testing. ContiPerf [2] and JUnitPerf [3] extend the JUnit test framework to measure software performance and scalability and report the performance regression on the unit test level. Developers encode the performance threshold in a test case and performance regression is identfied by failed test cases. However, neither of them reason about the program and identify the root cause of the regression. Consequently, developers still have to inspect the source code manually to locate the code responsible for the performance difference.

Continuous Peformance Testing. J. Davison de St. Germain et. al. [4] present an approach to track performance on parallel software projects, by integrating specific performance benchmarks [5] in the Uintah Computational Framework (UCF). In particular, they continuously run the performance test case during development and provide a historical portfolio of the evolution of application performance. While this approach provides important insights, it is limited to UCF and parallel software and cannot be easily applied to a broader range of products. One of the prior works in this area is related to the measurement and identification of performance failures [6] by Juan Pablo et. al. In this paper, the authors try to identify the inconsistencies between different execution results of the profiler on the same application version. The Aim of Pablo's paper is to solve the caching issue of the profiler output. As the output of the profiler is ad hoc and inconsistent it is very difficult to cache the result. They have proposed a new technique to solve the problem of caching. This paper is inclined more towards improving the output of the profiler rather than actually studying obscure performance issues embedded as the software evolves. Our technique on the other hand analyzes the different versions of the application to identify the performance issues and clearly points out the exact functions that have caused the performance regression.

*Performance Rootcause Analysis.*Christoph Heger et. al. in [7] propose a technique to identify and present the root cause analysis for performance regression from the previous version of the code. In this context the authors propose an approach named PRCA (Performance regression Root Cause Analysis) for identifying the root cause for performance regression. In the PRCA, the author calculate performance based on junit test execution time. Identifying performance regression based on Junit execution is not a good approach as Junit test cases are written to test the functionality rather than performance of the code.PerfDiff on the other hand identifies the performance regression using a profiler, which more reliable in identifying the performance issues.

Another relevant work that deals with monitoring and localization of performance issues is [8] by Jens Ehlers et. al. In this work the authors propose a rule-based approach for monitoring the performance anomalies at run time. In this attempt they develop a plug-in as an extension to Kiekers framework and define the monitoring rules using Object Constraint Language (OCL). These rules represent responsiveness (a performance attribute), an anomaly which usually varies in value during the runtime. They have implemented the feature of self - adpatation by continuous evaluations of the monitoring rules. The self adaptation has been implemented based on Eclipse Modeling Framework (EMF). Although we share the idea of performance issue localization with Jens Ehlers et. al., our approach is very simple and efficient in terms of performance issue localization. Moreover, to use Jens Ehlers approach the user should be well-versed to use the complex frameworks (Kiekers and EMF frameworks). On the other hand to use our tool the user just needs to know how to commit the code to GitHub.

The work [9] done by Andrea Adamoli et. al is related to identifying the performance issues in terms of the responsiveness of the system. This work is similar to our current work in terms of identifying performance issues. The approach employed by Andrea et. al is based on CCT (Calling Context Tree) visualization. In this work the authors build a framework called FlyBy that is integrated to work with the application. Although we share the idea of identifying performance issues, there are many difference in terms of the approach. One important difference is that Andrea et. al.'s tool takes in complaints from the user when the user presses a complaint button on the application and collects the application profile at that time and analyzes the cause for the reduction in responsiveness of the application. When compared to Andrea et. al's work which is post production analysis, our approach identifies the performance issues at a very early stage of development. When our approach is employed, customer satisfaction would be higher for two reasons. One, the user can enjoys using the application and is not involved in reporting performance issues. Second, our approach detects the issues at an early stage when there is no additional cost overhead involved due the performance defects.

*Record Performance Results.*A very closely related work that shares almost the same idea of identifying performance anomalies whenever a user commits a new revision to version control system is [10] by Nagy Mostafa et. al. In this work , the authors purpose a new revision control system with PARCS service (Performance - Aware Revision Control System), a service that provides feedback to the developers as how a change that they have committed affected the behaviour and performance of the overall application. They have employed Calling Context Tree (CCT) profiling for identifying the performance differences of the two revisions of the same application. We share a similar idea of enabling the revision control system to be aware of the performance issues. Nagy Mosftafa's approach reasons about performance regression in terms of method additions and deletions. Moreover, it is not clear from their evaluation results what parameters of performance issues are being measured. In our approach, we give concrete execution times for the changed functions . Also, we record the difference in the repository along with the code, to be examined.

*Related Work In Performance Regression.*Various projects design specific regression tests that are geared towards performance. For instance, the Chromium project, creates regression tests with a threshold output which determines if for a particular value, the new changes have regressed in performance. The Zoom system-wide profiler provides hints to programmers about possibly slow operations. Altman et al. presented a tool, WAIT, to diagnose root cause of idle times in server applications [11]. Jovic et. al. proposed lag hunting, an approach to identify perceptible performance bugs by monitoring the behavior of applications deployed in wild

[12]. However, this approach focuses on post-development analysis and involves lots of user interaction. To some extent, this tool wouldn't help a lot during the development process.

In addition to the work aforementioned, a variety of other work has been done in performance prediction and analysis of both low-levl (C) and high-level (C++) programs in embedded systems and parallel applications [13, 14, 15, 16, 17, 18, 19]. However, the tool we propose is meant to be integrated into the development process and execute in the background everytime a user commits code, so that performance bugs can be identified more rapidly during code review.

3. METHODOLOGY

3.1 Approach

A variety of performance analysis tools exist for both serial and parallel programs, with varied degrees of sophistication and intrusiveness. Our current primary candidates are gprof and Google performance tools since they are both are freely available, easy to inject at compile time and add minimal overhead to execution time. Both profilers use sampling in order to capture the location of the program counter at a constant interval. They both use signals in order to sample, but Google Performance Tools has the advantage of accomodating for time spent in libraries that are not directly profiled. On the otherhand, gprof provides a text representation of a call graph, where as Google Performance Tools does not. Regardless of the profiler used, the profiled test is run against the new (or pending) version and the previous version of a project to retrieve performance data.

On most large projects, software changes are first run independently on a build server for regression testing before upstreaming to the main project repository. The goal of our tool is to operate at the point after a developer makes a commit to the build server, providing a visualization of the performance differences as part of the github user interface between the new version and previous versions of the code.

For the purposes of the mid-point evaluation (Phase 1) and demonstration, we pulled two versions of performance sensitive C code we wrote from a repository to compare.

In the first phase, we prepared the inputs to a script that ran the profiling tool and extracted function level information from the output. First, a working directory and a cloned version from a Git repository was gathered, one serving as the new version and one as the previous version to compare to. Using a customized makefile specifying which tests are to be run, the two versions were compiled and a list of test binaries was output for each. Next, we obtained a list of functions that have been modified between the current and previous versions of the codes by parsing the source code of both. Since in general, developers may use functions written by someone else in their own functions, oftentimes they may forget or be unaware of their code's dependencies. For this reason, we also allowed the user to input a list of functions of which they specifically want to see the performance metrics, even if these functions were not changed syntactically. This list combined with the list of modified functions, along with the test binaries for the current and previous versions is the input to our profiling script. Next we ran the chosen profiling tool with the test binaries of the current and previous versions, focusing on extracting the performance data of the functions in the input lists. As part of Phase 1, we measured CPU execution time as the performance metric, normalized by the execution time of a base function. We then simply outputted a list of function names corresponding to their speedup factors (ratio of new to old execution times) to the console.

In the second phase, we have improvised our approach in terms of usabililty and visualization. In terms of usability, we have automated the process of installing our tool as pre-commit hook script and automated the configuration of the post-push wrapper. After installing, the user specifies the GitHub code base url, current version repo token, path to uncommitted code, GitHub User ID in configuration file (perfDiff-config.cfg). We have improvised the tool to customize git version control system. With PerfDiff we have made the git version system to be "Performance - Aware". Once the user install PerfDiff, configures the configuration file and then commits the code to the git repository, the pre-commit hook triggers PerfDiff and collects and serializes the performance regression output, in addition to outputting to the console. For the visualization component, we have created a post-push wrapper that actually posts the performance regression output as inline comments on the GitHub repository.

Our software operates on the notion that requests are made with compiled versions of a particular project. We demonstrate this functionality by providing a script that automatically trigger the PerfDiff tool in the background when the user commits the code. To extend on our performance differencing tool in the future, we plan to analyze the performance regression with a deeper analysis by using call trace comparisons between the two versions. Currently we display performance regression of functions that changed between revisions. By following a call trace, we can better determine the cause of the performance change. Apart from deeper analysis, other aspects that we plan to expand upon is to evaluate on a larger test data set.

3.2 Implementation

PerfDiff is composed of a series of automation scripts and parsers that retrieve, compile, execute, and present performance data output. First, our tool pulls a specified revision of the target software and compares the files with those in the users current working directory. Using CDiffer, we generate a list of changed functions to add to a sensitivity list which is used later in the process. We use a custom make file to build both versions of projects with compile time flags that allow the profiling tool to hook into the code. We feed both binaries to the profiling tool (gprof) and parse the outputs. Based on the sensitivity list of functions we generated before, we display the relevent performance differences in the functions where code was modified. Our software involved integrating multiple software suites including python, pycparser, gprof, make, git, and the target C code our software will analyze.

For the presentation we will demonstrate the installation process, running a git commit, and the resulting output and git annotations generated when compared against sample older versions in the repository.

3.2.1 PerfDiff Installation

We have integrated the PerfDiff tool to run as a wrapper around the git functionality. Once the user installs the

PerfDiff tool and runs the install script, the tool automatically configure the pre-commit hook and also configures a post-push wrapper script for git push command. The user has to configure various parameters in the PerfDiff-config.cfg file.

3.2.2 Pre-Commit Hook

We have integrated a pre-commit hook to trigger PerfDiff tool automatically when ever the user commits a new version of the code to the repository. The pre-commit hook runs the tool and serializes the performance data, which is later used by the post-push wrapper.

The pre-commit hook script is summarized in the following python code.

```
# read config file
config.read('/Config/PerfDiff.cfg')

#extract the github repository link, retreive the
     acccess token, revision number, curent revision
     number

# generate binaries from git
create_bin(src_dir, tmp_dir, version)

# extract user requested functions
reqFuncs = extractReqFuncs(requestedFuncsFile)

# find the changed functions
chgFuncs = runCDiff(src_dir, tmp_dir)

# run profiler on each binary
src = runProfiler(...binSrc..., chgFuncs, reqFuncs)
tmp = runProfiler(...binTmp..., chgFuncs, reqFuncs)

# compute the speed (src,tmp)
speedFuncs = computeSpeedUp(src,tmp)

# serializing the output
pickle.dump(listOfEntries, file('entries','w'))
```

3.2.3 Build Module

The build module configures the two target program directories and compiles the source files with the proper flags so that the profiler can hook into the binaries.

The following are inputs to the module:

- local directory: contains the source the user is working on.
- temporary directory: contains another version of the same source the user is working on.
- makefile: contains the specific macros and build configuration for the target project

3.2.4 CDiffer Module

CDiffer uses coarse-grained differencing approach, providing functions that have evolved from the previous version of the program. It utilizes pycparser to parse C source code to abstract syntact tree (AST), in order to identify functions and their coordinates(e.g.line numbers) in the program. Currently, code changes are computed by Linux Diff, a tool for finding textual differences between two files in Linux. However, it can be extended by using some syntactic or semantic differencing algorithms as a part of further work. Changed functions are computed by comparing change range with function coordinates. Any function that contains at least one change is considered as changed. Changed functions are used later for reasoning performance evolution in Profiler Module.

Algorithm 1: CDIFFER finds the changed methods

Input: $prev \leftarrow$ git path of previous revision $head \leftarrow$ git path of head revision

Output: A list of tuples $\langle\langle func, line, file\rangle \ldots,\rangle$

```
changed_funcs ← []
while there are files in prev and head do
    oldFile ← c file name in prev
    newFile ← c file name in head
    if oldFile = newFile then
        call diff
        ranges ← line numbers of changes
        for function in newFile do
            range ← start and end position of function
            if range in ranges then
                changed_funcs ←
                ⟨ranges, function, newFile⟩
return changed_funcs
```

3.2.5 Profiler Module

The profiler module is responsible for configuring the environment for profiler execution, running the profiler on a specific binary, and parsing the output of the profiler to extract relevant performance information. It is also responsible for populating a list containing annotation information that is eventually passed to the post-push script.

In order to normalize CPU speeds between executions of the profiler, a *base()* function that is identical between version is always executed.

The following are inputs to the module:

- directories: two directories with varying versions of the same source code that we want to profile.
- changed functions: a list of functions that changed between versions
- path and line numbers: these are needed in order for the post-push hook to know where to add the annotation.

The profiler is run on the executable contained in the directory specified as an input. The speed of each function is computed based on the sample rate, and a list of tuples *(function, speed, path, line number, list of changed functions)* is returned.

3.2.6 Result Viewer Module

The result viewer module wraps the performance data generated by the profiler module so that we can submit a POST request and show the performance changes inline with diff on Github. The Github API allows users to post inline comments with diffs. PerfDiff leverages this API and appends performance data after a push is done to the global repository. The module parses an output file from the profiler and deserializes the information to post it to Github.

On GitHub the user can choose to view a diff and there is a checkbox option that will allow them to view the inline performance data. This provides immediate feedback on the performance difference between the two versions. This data can be extended upon to view trends between versions overtime to show if performance degrades or improves over time.

3.3 System Architecture

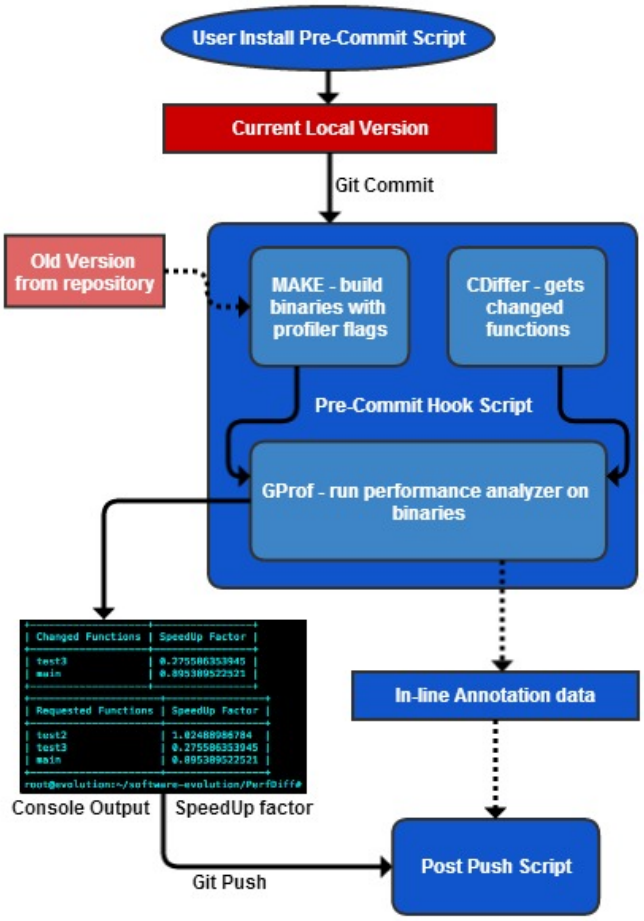

Figure 1: System Architecture Overview

3.4 User Interface

The preliminary terminal based user interface makes use of the PrettyTable module in python in order to display two tables to the user: 1) The changed functions between the two versions, 2) The functions specifically requested by the user.

Figure 2 shows a sample execution of our tool. In this case, *test3* and *main* were changed between the two versions of the program. The user also requested to be notified regarding performance of *test2*. From this simple output we can deduce that the performance of *test2* did not change. However, *test3* ran over 3 times more slowly (changing by a speedup factor of 0.27).

```
+--------------------+--------------------+
| Changed Functions  | SpeedUp Factor     |
+--------------------+--------------------+
| test3              | 0.275586353945     |
| main               | 0.895389522521     |
+--------------------+--------------------+

+--------------------+--------------------+
| Requested Functions | SpeedUp Factor    |
+--------------------+--------------------+
| test2              | 1.02488986784      |
| test3              | 0.275586353945     |
| main               | 0.895389522521     |
+--------------------+--------------------+
root@evolution:~/software-evolution/PerfDiff#
```

Figure 2: Sample Terminal Output of PerfDiff

4. EVALUATION

4.1 Experiments

In order to evaluate our performance differencing tool, we created a sample code base that performs line of sight calculations on a hexagonal map of an arbitrary size. The first version uses the even-r offset coordinate system, which is a traditional two-axis coordinate system with the origin in the upper left-hand corner and the axes aligned horizontally and vertically. The second version uses the axial coordinate system, which has a diagonal vertical axis. The final version uses the cubic coordinate system, which adds a third axis so each face of the hexagon is aligned with one of the axes. Our hypothetical developers are experimenting with these different coordinate systems to determine the tradeoff between computational efficiency and readability.

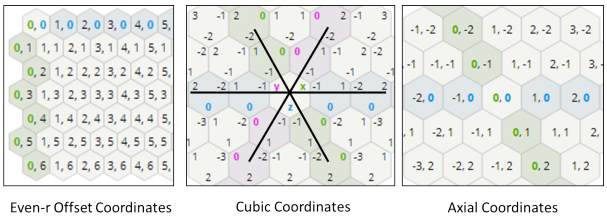

Figure 3: Graphical Comparison of Hex Coordinate Systems. [20]

There are four test cases to simulate a range of scenarios for applying these calculations. Test1 and Test2 calculate the distance, bearing, and hexes on a line between two points. This could be used to calculate if there is anything blocking the line of sight between the observer (origin) and the target point. These test cases differ in the method used to determine the target point: Test1 iterates over every single hex in the map, whereas Test2 generates the target points randomly. Test2 generates the same number of points as there are hexes so the run lengths should be similar.

Test3 only performs the distance and bearing calculations between two points. This simulates if the target is outside of the maximum view range of the observer's sensors, which makes it pointless to calculate if there are any obstructions between the two points. These points are generated randomly, and since this function takes considerably less time

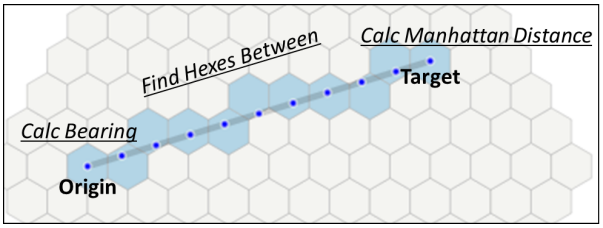

Figure 4: Line of Sight Calculation

to execute than the full line of sight calculation, Test3 is run on 250 times as many target points as there are in Test2. This numer was chosen so that Test3 takes approximately the same length to run as Test1 or Test2.

Test4 converts the origin and a random target coordinate point into even-r offset coordinates. This represents the computational burden for converting the more efficient cubic or axial coordinate systems into a readable format. In order to run as long as the other test cases, 1250 times as many target coordinates are generated as compared to test case two.

	Test 1	Test 2	Test 3	Test 4
Function Call	ScanLOSCalc	RandLOSCalc	RandLOSCalcMin	RandDisplayCoord
Library Calls	- Distance - Bearing - Hexes Between	- Distance - Bearing - Hexes Between	- Distance - Bearing	- Converts coordinates to Offset for readable display
Description	Calculates results from origin to every point on the map	Calculates results from origin to a random point.	Calculates results from origin to a random point.	Calculates results for origin and random point
Iterations	X_SIZE * Y_SIZE	X_SIZE * Y_SIZE	X_SIZE * Y_SIZE * 400	X_SIZE * Y_SIZE * 1250

Figure 5: Comparison of Test Cases

To evaluate PerfDiff, we compared the three versions of the code (offset, then axial, and finally cubic coordinate systems) to ensure that the program properly displays the changed and specified functions and that the profiler results are accurately displayed. We ran each coordinate system twenty times to determine the mean and standard deviation for each system.

We also determined the impact of background processes or multi-tasking on the results from the profiler and evaluate a normalizing function that could reduce this imact. In order to do this, we ran the profiler on the offset test cases without any backround processes, and then for comparison we ran the same test with several background programs running (virus scanner and streaming media while using Microsoft Excel). We generated these test sets by averaging the results of twenty runs each, and from this we extrapolated a third one data set containing a 50-50 mix of the two. We then use a base function (which remains constant across all three versions) to try and normalize the results for each run and use the normalized performance data to compare the different versions of the code.

Finally, we conducted a series of tests to determine how the the duration of each test case impacts the fidelity and variance of the results. To determine this, we compared the results from test runs ranging in length from 0.2 seconds to roughly 9 seconds by increasing x-dimension of the map in increments of 100, starting at 200 and ending at 1400, while holding y-size constant at 100. Each map size was tested ten times per coordinate system and we compared the mean and standard deviation of the different coordinate systems to determine how much shorter test cases will impact the results. The profiler samples by default 100 times every second, so these tests will show at what point the sampling rate should be increased to generate accurate results.

4.2 Results

4.2.1 *Relative Performance Between Versions*

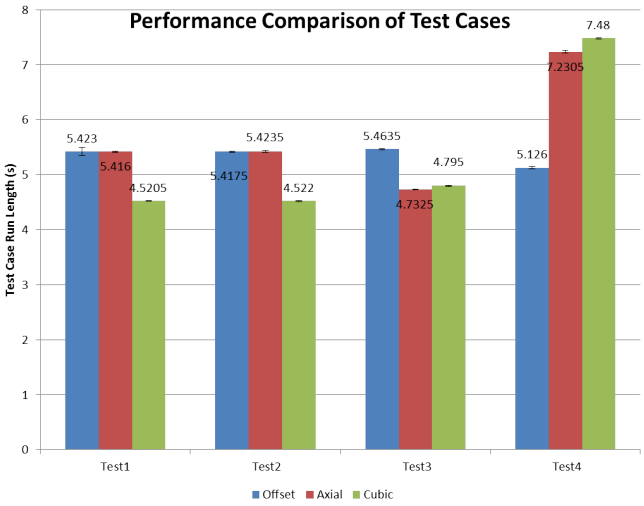

Figure 6: Comparison of Test Results

The results from the test cases are shown in absolute terms of CPU usage in order to make the data easy to read and compare across all three versions. Test case performance is shown in seconds, which is obtained by taking the output of the profiler and dividing the number of samples in a function by the sampling rate. The performance of Test1 and Test2 were similar, both for offset and axial systems. An improvement of nearly 20% is achieved when using the cubic coordinate system. Test3 shows that both axial and cubic systems show an approximate 15% improvement, with little difference between axial or cubic performance. Finally, Test4 shows a 30% performance drop from offset to axial, but then only a small degredation of 3.4% going from axial to cubic. This is expected since no conversions need to be done to print coordinates in offset coordinates.

By showing the relative performance differences between versions of changed and specified functions, PerfDiff streamlines what would ordinarily take several steps to manually implement (run profiler and sift through the results, then manually compare their performance, or alternatively manually tag the program and then compare the results). Furthermore, by identifying changed functions that show a significant performance change that allows the added ability to detect when code changes adversely affect performance of other parts of the program.

4.2.2 *Impact of Other Processes on Results*

Our evaluation of system load impact on the results show that, predictably, running the profiler with background processes creates less consistent results, especially in the 50-50

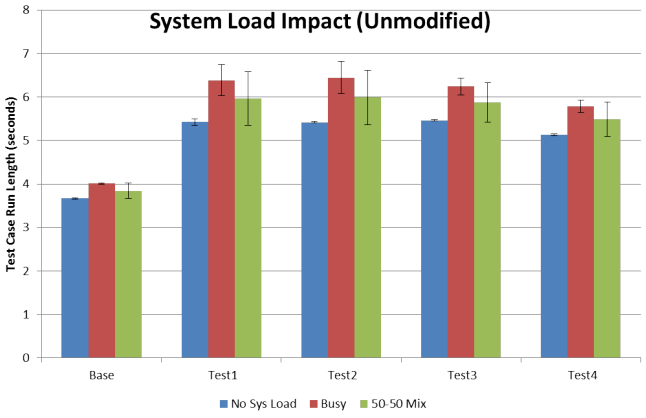

Figure 7: Impact of System Load on Unmodified Results

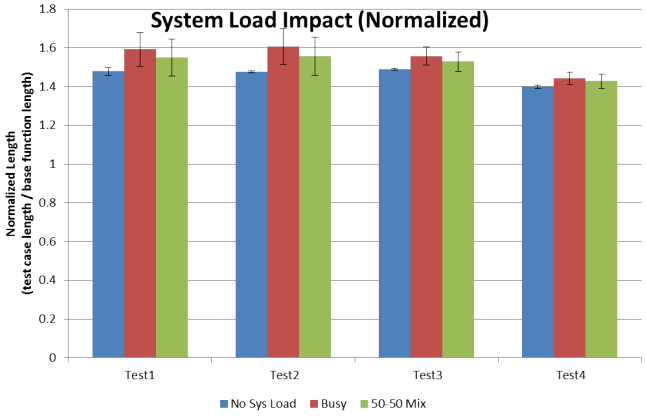

Figure 8: Impact of System Load on Normalized Results

mix of the results. Once we apply normalization to the test results by dividing the number of profiling samples in the test case by those in the base function, this successfully limits the added deviation caused by an inconsistent system load. Even though it would clearly be better to simply run the profiler without any background processes, normalizing with the base function mitigates the impact on the profiler results in case, e.g., a pre-scheduled virus scan starts in the middle of the test run.

4.2.3 Impact of Test Run Length

Finally, we assessed how the run length of the test cases impact how accurately PerfDiff records the relative performance between functions. Long run lengths (greater than three seconds) show diminishing returns for increased run length, and most test cases stabilized beyond the one second mark. Test cases running shorter than one second can have significant errors in accuracy, such as Test1 between versions 1 and 2. In this case, the relative performance of 84% for the 0.2 second run is significantly different from the 92.5% in the 7 second run).

Since the library functions being called in our test code all take a very short amount of time, test run length is determined by the number of iterations each function is called (in this case, based on the dimensions of the map). In these circumstances, test cases don't need to run for more than

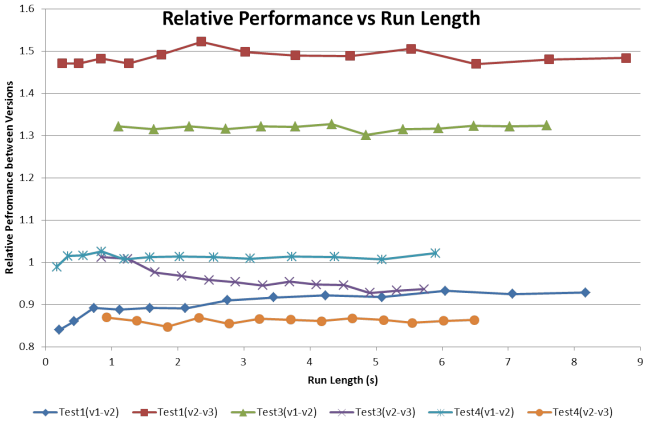

Figure 9: Run Length and Relative Performance

a few seconds, and if any test case runs for less than a second, the sampling rate of the profiler should be increased accordingly.

4.3 User Study

In order to show the usefulness of PerfDiff qualitatively, we conducted a very small scale user study. Student A (Brian) wrote the sample test / client code. Student B (Tomasz) was asked to use the tool to determine the location of performance bugs, and to answer some questions posed by Student A. The following questions were posed:

- Where does performance improve / degrade between these two versions of code?
- What information can you provide regarding the cause of the performance changes?

Since three versions of the code were written, Student B was asked to perform 2 comparisons: (1) Version 2 to Version 3, (2) Version 1 to Version 2. Since gprof is limited in accuracy and the tool's primary purpose is finding significant performance differences, our tool uses a threshold of 10 percent (the reasoning for this is explained further in the following section).

For comparison (1), Student B found that there were large performance degradations for three out of four tests. Student B was able to see that only a few functions that changed also had performance changes. These were all client functions, which led Student B to conclude that the performance bugs were introduced by using the library code incorrectly rather than the library code itself degrading.

Amongst these there was further variability in the amount of performance degradation. For example, ScanLosCalc degraded by a factor of 0.881, while RandLOSCalc degraded by a significantly worse factor of 0.439. Two other client functions degraded at rates similar to RandLOSCalc. Therefore, the tool is able to tell student B that a performance bug likely exists in one of these three functions. The library code functions did not execute long enough to show up as having performance differences, so the tool is unable to guide Student B. However, by pointing the user to these functions, he would be able to determine that a duplicate LOSCalc function call was added to the Version 3 line of sight calculation

function. Furthermore, the common performance degredation in test cases calling random functions could lead him to inspect the random number generation feature, as a new function was created to generate random coordinate pairs in the cubic coordinate system used in version 3.

For comparison (2), Student B found that test3() and test4() were the only functions that showed significant performance differences.

Student B found that performance for test3() improved by a factor of 1.320, where as performance for test4() degraded by 0.676. Unfortunately, the tool did not provide additional information regarding the API functions responsible for these differences. This is likely because the API functions themselves did not execute for long enough for the differences in their execution time to show up through gprof. However, since the test cases called these functions repeatedly, their execution time was long enough for gprof to notice.

This shows a limitation of the tool. Specifically, the tool is not suited for diagnosing small scale performance bugs, where a function takes a relatively large amount of time, but the absolute amount of time is still low. This means performance impacts will only be significant if it is called frequently.

4.4 Validation

PerfDiff aims to provide performance information comparing code revisions. To validate this functionality we had to consider what factors in our tool chain impacted the performance data output. The main source of variance in our tool is the user's system status. Since PerfDiff uses runtime profile infromation, it is subject to the real-time load factor on a users computer. If the user has background processes that have periodic increases in cpu usage, it will impact how PerfDiff measures performance regression. To adjust for these changes, we introduce a dummy function that is run as a control. The performance information gathered for the control version impacts the ratio that PerfDiff will produce for the actual tests run on the code base. PerfDiff also uses a threshold that ignores performance changes if it does not meet our significance criteria. The significance criteria is derived from the expected value and variance calculated from gprof. By running gprof multiple times on the same code base, we can view the range of performance data it provides and calculate an error threshold. Based on this threshold, we determined that performance difference less than 10% do not satisfy the significance criteria.

5. FUTURE WORK

There are a few areas in which we hope to focus our future efforts in extending PerfDiff. Firstly, we wanted to use the call graph to assist in finding the cause of performance changes. We could potentially use gprof to obtain a call graph of the project, then use this to identify dependencies between functions and better pinpoint the causes of performance changes. This allows the user to identify functions in which they are interested that may not have been explicitly modified, but whose performance might have been affected by other modified functions.

Second, we want to be able to better track performance changes over time. Our current permanent annotations in the version history display a speed up factor, or relative performance, when compared to the *previous* version only. We would like to be able to take this information and for a certain function, show its performance over its entire history.

Originally we had envisioned PerfDiff running on a build server, not on a developer's local machine. For the proof of concept and prototype we decided not to use this setup and just run it locally, but eventually we would want the tool to be run automatically whenever someone sent a new version to the build server. We also hope to run PerfDiff on a much larger base of performance sensitive code to examine its usefulness when there are many functions being profiled.

Lastly, PerfDiff could also be run in the background as a speculative tool even before a user does a git commit. This way PerfDiff could alert a programmer even quicker if they are making changes to code that would adversely affect performance.

6. REFERENCES

[1] Antonino Sabetta and Heiko Koziolek. Performance metrics in software design models. In *Dependability Metrics, Lecture Notes in Computer Science Volume 4909*, pages 219–225. Springer Berlin Heidelberg, 2008.

[2] Volker Bergmann IT Consulting. Contiperf, 2012.

[3] Mike Clark. Junitperf, 2012.

[4] J Davison de St Germain, Alan Morris, Steven G Parker, Allen D Malony, and Sameer Shende. Performance analysis integration in the uintah software development cycle. *International Journal of Parallel Programming*, 31(1):35–53, 2003.

[5] Lubomir Bulej, Tomas Kalibera, and Petr Tma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1):345–358, 2005.

[6] Juan Pablo Sandoval and Alexandre Bergel. Debugging performance failures. In *Proceedings of the 6th Workshop on Dynamic Languages and Applications*, page 2. ACM, 2012.

[7] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the ACM/SPEC international conference on International conference on performance engineering*, pages 27–38. ACM, 2013.

[8] Jens Ehlers, André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 197–200. ACM, 2011.

[9] Andrea Adamoli and Matthias Hauswirth. Trevis: a context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th international symposium on Software visualization*, pages 73–82. ACM, 2010.

[10] Nagy Mostafa and Chandra Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 162–171. ACM, 2009.

[11] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 739–753. ACM, 2010.

[12] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *ACM SIGPLAN Notices*, volume 46. ACM, 2011.

[13] Sri Hari Krishna Narayanan, B. Norris, and P.D. Hovland. Generating performance bounds from source code. In *Parallel Processing Workshops, 39th ICPPW*, pages 197–206. ICPPW, 2010.

[14] Yong-Yoon Cho, Jong-Bae Moon, and Young-Chul Kim. A system for performance evaluation of embedded software. *Transactions on Engineering, Computing and Technology*, 1:1305–5313, 2004.

[15] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation*, pages 441–452. PLDI, 2009.

[16] Markus Geimer, Felix Wolf, Brian J. N Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 301–312. European PVM/MPI Users' Group Meeting, Springer Berlin Heidelberg, 2006.

[17] K.W Rong Ge; Xizhou Feng; Shuaiwen Song; Hung-Ching Chang; Dong Li; Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(5):658–671, 2010.

[18] Robert Bell, Allen D. Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003 Parallel Processing*, pages 17–26. Euro-Par 2003 Parallel Processing, Springer Berlin Heidelberg, 2003.

[19] S Balsamo, A Di Marco, P Inverardi, and M Simeoni. Model-based performance prediction in software development: a survey. In *IEEE Transactions on Software Engineering*, volume 30, pages 295–310. IEEE Computer Society, 2004.

[20] Amit Patel. Hexagonal grids from red blob games. http://www.redblobgames.com/grids/hexagons/, 2013.