Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **26**:1193–1212
Published online 17 July 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1657

Detecting and analyzing I/O performance regressions

C. Bezemer*,†, E. Milon, A. Zaidman and J. Pouwelse

Delft University of Technology, Delft, the Netherlands

SUMMARY

Regression testing can be performed by reexecuting a test suite on different software versions and comparing the outcome. For functional testing, the outcome of such tests is either pass (correct behavior) or fail (incorrect behavior). For nonfunctional testing, such as performance testing, this is more challenging as correct and incorrect are not clearly defined concepts for these types of testing.

In this paper, we present an approach for detecting and analyzing input/output (I/O) performance regressions. Our method is supplemental to existing profilers, and its goal is to analyze the effect of source code changes on the performance of a system. In this paper, we focus on analyzing the amount of I/O writes being performed. The open source implementation of our approach, SPECTRAPERF, is available for download. We evaluate our approach in a field user study on Tribler, an open source peer-to-peer client and its decentralized solution for synchronizing messages, Dispersy. In this evaluation, we show that our approach can guide the performance optimization process, as it helps developers to find performance bottlenecks on the one hand and, on the other, allows them to validate the effect of performance optimizations. In addition, we perform a feasibility study on Django, the most popular Python project on Github, to demonstrate our applicability on other projects. Copyright © 2014 John Wiley & Sons, Ltd.

Received 14 November 2013; Revised 22 May 2014; Accepted 30 May 2014

KEY WORDS: performance regressions; performance analysis; performance optimization

1. INTRODUCTION

Regression testing is performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program [1]. It can be performedby reexecuting a test suite on different software versions and comparing the test suite outcome.

For functional testing, the functionality of a program is either correct or incorrect. Hence, the outcome of such tests is either pass or fail. For nonfunctional testing, correct and incorrect are not clearly defined concepts [2], making regression testing even more challenging. An example of nonfunctional testing is performance testing. Two possible reasons for performance testing are the following:

1. To ensure that the software behaves within the limits specified in a service-level agreement
2. To find bottlenecks or validate performance optimizations

Service-level agreement limits are often specified as hard thresholds for execution/response time, that is, the maximum number of milliseconds a certain task may take. The main reason for this is that execution time influences the user-perceived performance the most [3]. For performance optimizations, such a limit is not precisely defined but follows from a comparison with the previous software version instead, as the goal is to make a task perform as fast or efficient as possible.

*Correspondence to: C. Bezemer, Delft University of Technology, Delft, the Netherlands.
†E-mail: c.bezemer@tudelft.nl

Hence, we are interested in finding out whether a specific version of an application runs faster or more efficiently than its predecessor.

As a result, including performance tests in the regression testing process may provide opportunities for performance optimization. In fact, in this paper, we will show that the outcome of these tests can guide the optimization process.

Performance optimization can be performed on various metrics. Execution time, which is the most well known, can be analyzed using traditional profilers. Other examples of metrics that can be optimized are the amount of input/output (I/O), memory usage, and CPU usage. These metrics are difficult to analyze for software written in higher level languages, such as Python, because of the lack of tools. Hence, the understanding of how software written in such languages behaves regarding these metrics is often low [4]. In addition, understanding the performance of a system in general is difficult because it is affected by every aspect of the design, code and execution environment [5].

In this paper, we propose a method that helps performance experts understand how the performance, focusing on the I/O writes metric mentioned in the previous texts, changes over the different versions of their software. Performance can be monitored on various levels, for example, on the system level [6] and on the function level. Profilers can monitor performance on the function level. Our method is supplemental to existing profilers in the way that it uses the output of these tools to analyze the effect of source code changes on the performance of a system. We achieve this by monitoring the execution of a specific test for two versions of an application and comparing the results. The result of our approach is a report that helps a performance expert to

1. Understand the impact on write performance of the changes made to the software on a function-level granularity
2. Identify potential optimization opportunities by finding regressions or validate fixes

We evaluate our approach in a field user study on a decentralized peer-to-peer (P2P) BitTorrent client, Tribler [7]. In the first part of our study, we analyze the performance history of a component in Tribler by analyzing its unit test suite. In the second part, we analyze the effect of nondeterminism on our approach, by analyzing a 10-min run of Tribler in the wild. Finally, we ensure the external validity of our approach in a feasibility study on Django, a popular Python web framework.

The outline of this paper is as follows. In the next section, we first give a motivational example for our approach. In Section 3, we present our problem statement.

In Section 4, we introduce our approach for detecting and analyzing I/O performance regressions. Section 5 details the implementation of our approach, called SPECTRAPERF. In Section 6, we discuss the setup and results of our user study on Tribler and Dispersy. Section 7 deals with the results of our feasibility study on Django. We discuss these results and the limitations of our approach in Section 8. In Section 9, we discuss related work. We conclude our work in Section 10.

2. MOTIVATIONAL EXAMPLE

In a database system, some queries cause the creation of a temporary table.[‡] The creation of such a table is often performed silently by the database system itself but may be intensive in terms of I/O usage. For example, SQLite creates a temporary file to store the table in. Finding out which function causes the temporary table creation can help reduce the I/O footprint of an application. Because I/O takes time, we can detect this behavior using a traditional profiler, which is based on execution time. However, there is no information available about whether the function resulted in the creation of a temporary table or that the high execution time was caused by something else. This makes the issue hard to diagnose and optimize. In addition, if a developer has found the cause of the temporary table generation, a fix is difficult to validate because of the same reason: It is difficult to verify whether the temporary file was indeed not created or that it was simply created and deleted. Using an approach that can automate this

[‡]For example, for SQLite, http://www.sqlite.org/tempfiles.html

process, we can see if a function has started generating temporary tables since the previous version. Then, after fixing it, we can validate if our optimization had the desired effect.

3. PROBLEM STATEMENT

By including performance testing in the regression testing process, developers can get feedback about the way their changes to the code impact the performance of the application. This feedback can be used to as follows :(i) be warned of undesired negative effects or (ii) validate the positive effect of a performance bug fix. To give this feedback, we must do the following:

1. Define which metrics that we want to analyze and combine this set of metrics into a *performance profile*, which describes the performance behavior of a revision
2. Generate such a performance profile for every source code revision
3. Compare the most recent profile with the profile(s) of the preceding revision(s)
4. Analyze which source code change(s) caused the change(s) in performance

In this paper, we focus on the following research question:

Main RQ: How can we guide the performance optimization process by doing performance regression tests?

To answer this research question, we divide it into the subquestions discussed in the remainder of this section.

RQ 1: How can we monitor performance data and generate a comparable profile out of these data?

Depending on which metric that we want to analyze, we must find a suitable monitor to record performance data. Ideally, we want to be able to monitor without needing to change the source code of the application. An additional challenge is that an application may use libraries written in different programing languages, making it more difficult to get fine-grained information about, for example, I/O.

A challenge is formed by the fact that monitoring the same test twice may result in slightly different performance profiles because of variations in, for example, data contents and current memory usage [8] or other applications running on the same machine. As such, we must devise a method for comparing these profiles:

RQ 2: How can we compare the generated performance profiles?

Finally, we must be able to analyze the differences between profiles and report on the functions most likely to cause the change in performance:

RQ 3: How can we analyze and report on the differences between profiles?

In this paper, we investigate an approach that helps us to detect and analyze performance regressions. In this study, we focus on detecting and analyzing performance regression caused by write I/O. We expect that our approach can be adapted to work for other performance metrics, which we will verify in future work.

4. APPROACH

The goal of our approach is to analyze the effect of source code changes on the performance of a system. Ideally, we would like to be able to generate a report explaining per function how much a performance metric changed, compared with the previous source code revision. In this section, we explain our approach for generating such a report. The idea of our approach is that we summarize the behavior of an application during the execution of a test execution in a *profile*. After an update, we compare the behavior of our application during the execution of the same test using that profile.

4.1. Profile generation

To be able to report on a function-level granularity, we must also monitor data on this granularity. Therefore, we first automatically instrument (refer to Section 5) all functions in our application that perform writes. The instrumentation code writes an entry to the log for every write action, containing the number of bytes written, the name of the function and the location of the file being written to.

Second, we let the instrumented code execute a test, which generates a log of all write actions made during that execution. This test can be any existing, repeatable test (suite), for example, a unit test or integration test suite. The write actions made to the log are filtered out from this process.

To lessen the effect of variation within the program execution [8], for example, because of data content and current memory usage, we execute the test several times for each revision and combine the logged data into a performance profile. The number of times the test must be executed to get an accurate profile is defined by a trade-off between accuracy and test execution time. Ideally, we would like to run the test many times to get a more precise profile, but this may be impractical, depending on the execution time. A profile is generated by doing the following for every function:

- Calculate the average number of bytes that a function writes per call during a test execution (hence, divide the total number of bytes written by that function during the test execution by the total number of calls to that function during the test execution)
- For every test execution, define the highest and lowest values for this average number of bytes written per call as the accepted range for that revision for that function

Table I demonstrates this idea. The profile can be read as follows: 'During revision 1, `flushToDatabase()` wrote an average of 900 to 1500 bytes per call and `generateReport()` wrote an average of 1200 to 1604 bytes per call'.

4.2. Profile analysis

In order to assess the changes in performance of a revision, we compare its profile with the profile of the previous revision. While this can be performed manually, this is a tedious process and prone to mistakes. We propose to automate the comparison using a method inspired by spectrum-based fault localization (SFL) [9]. SFL is a technique that closely resembles the human diagnosis process, making the diagnosis easy to interpret.

Another advantage of automating this comparison is that we can use the technique in automated testing environments, such as continuous integration environments. To the best of our knowledge, we are the first to propose a method that is inspired by spectrum-based analysis for performance regression detection.

For every test execution t_i, we record the write data as described in Section 4.1. We verify for every function whether the recorded average number of bytes written falls in (1) or outside (0) the accepted range of the profile of the previous revision. As a result, we obtain a binary vector in which every row represents a function. If we place those vectors next to each other, we get a matrix. Table II shows sample data and the resulting matrix for three test executions t_i, after comparing them with the profile of Table I. We use three executions here for brevity, but this may be any number. As an illustration, we describe the formation of the (t_0) vector in Table II:

1. The value monitored for `flushToDatabase()` at t_0 is 1000. This value falls inside the accepted range (900–1500) for this function. Hence, the first value of t_0 is 1.

Table I. Illustration of the profile generation idea.

Revision 1	Average number of bytes written per call					Profile
Function	t_0	t_1	t_2	t_3	t_4	
flushToDatabase()	900	1000	1200	1100	1500	[900–1500]
generateReport()	1200	1500	1359	1604	1300	[1200–1604]

Table II. Illustration of profile comparison.

Revision 2	Average number of bytes written			Matrix			SC
Function	t_0	t_1	t_2	(t_0)	(t_1)	(t_2)	
flushToDatabase()	1000	1200	1100	1	1	1	1
generateReport()	2200	2000	1600	0	0	1	0.58
writeCache()	10 000	12 000	8000	0	0	0	0
Output vector				1	1	1	

2. The value monitored for `generateReport()` at t_0 is 2200. This value falls outside the accepted range (1200–1604) for this function. Hence, the second value of t_0 is 0.
3. The value monitored for `writeCache()` at t_0 is 10 000. This function does not have an accepted range defined in the profile. Hence, the third value of t_0 is 0.

Repeating this process for t_1 and t_2 results in vectors t_1 and t_2.

The analysis step now works as follows. When performance did not change after the source code was updated, all monitored values for all functions should fall into the accepted ranges of the profile of the previous revision. For three test executions, this is represented by the row [1 1 1] for every function. Any deviations from this mean that the average number of bytes written for that function was higher or lower than the accepted range. By calculating the similarity coefficient (SC) for each row and the 'ideal' vector [1 1 1], we can see whether the average number of bytes written for that function has changed (SC close to 0) or that it is similar to the previous profile (SC close to 1). As the SC, we chose to use Ochiai following advice from literature [10]. The Ochiai SC for two binary vectors v_1 and v_2 is defined as

$$SC = \sqrt{\frac{a}{a+b} * \frac{a}{a+c}} \tag{1}$$

with a as the number of positions occupied by 1 in both vectors, b as the number of positions occupied by 1 in v_1 and by 0 in v_2, and c as the number of positions occupied by 1 in v_2 and by 0 in v_1.

Using the SC, we can make a ranking of the functions most likely to have been affected by the update. When all SCs are close or equal to 1, the average number of bytes written did not change for any function after the update. The functions with SC closer to 0 are likely to have been affected by the update. As an illustration, we demonstrate how SC is calculated for `generateReport()`:

$$\begin{gathered} v_1 = [0\,0\,1] \\ v_2 = [1\,1\,1] \\ a = 1 \\ b = 0 \\ c = 2 \\ SC = \sqrt{\frac{1}{3}} = 0.58 \end{gathered}$$

In Table II, from the SC column, we can conclude that the performance of the `generateReport()` and `writeCache()` functions was likely to have been affected by the changes made for revision 2.

While the SC allows us to find *which* functions were affected by the update, it does not tell us *how* they were affected. For example, we cannot see if `writeCache()` started doing I/O in this version or that the amount of I/O increased or decreased. Therefore, we append the report with the average number of bytes that the monitored values were outside the accepted range (`Impact`). We also display the average number of calls and the `TotalImpact`, which is calculated by the average number of calls to that function multiplied with `Impact`. This allows us to see if the performance decreased or increased and by how much. In addition, we display the difference of the highest and lowest value in the range (`RangeDiff`). The goal of this is to help the performance expert understand the ranking better. For example, when a monitored value is 100 bytes outside the accepted range, there is a difference whether the range difference is small (e.g., 50 B) or larger (e.g., 50 KB). Additionally, we display the number of test executions out of the total number of test executions for this revision during which this function wrote bytes. This is important to know, as a function does not necessarily perform I/O in all executions. For example, an error log function may be triggered in only a few of the executions. A final extension that we make to our report is that we collect data for a complete stack trace instead of a single function.

We do this because of the following: (i) the behavior of a function may be defined by the origin from which it was called (e.g., a database `commit()`) and (ii) this makes the optimization process easier, as we have a preciser description of the function behavior.

Summarizing, the final report of our analysis contains a ranking of stack traces. In this ranking, the highest ranks are assigned to the traces of which the write behavior most likely has changed because of the source code changes in this revision. The ranking is made based on the SC (low to high) and the `TotalImpact` (high to low). In this way, the stack traces that were impacted the most, and were outside the accepted range in most test executions, are ranked on top. These stack traces are the most likely to represent performance regressions.

Table III shows the extended report. Throughout this paper, we will refer to this type of report as the *similarity report* for a revision.

From the similarity report, we can see that the average number of bytes written by `generateReport()` has increased relatively a lot compared with revision 1: The value for `Impact` is larger than the difference of the range in the profile. However, as SC and `TotalImpact` indicate, this was not the case for all test executions, and the average total impact was low. Additionally, we can immediately see from this report that `writeCache()` was either added to the code or started doing I/O compared with the previous version, as there was no accepted range defined for that function. In this case, `Impact` represents the average number of bytes written by that function. We can also see that the `TotalImpact` of the additional write traffic is 5 MB, which may be high or low, depending on the test suite and the type of application.

5. IMPLEMENTATION

In this section, we present the implementation of our approach called SPECTRAPERF. SPECTRAPERF is part of the open-source experiment runner framework Gumby[†] and is available for download from the Gumby repository. Our implementation consists of two parts, the data collection and the data processing part.

```
(begin)
=> python.function.entry
  => syscall.open.entry
  <= syscall.open.return
  => syscall.write.entry
  <= syscall.write.return
<= python.function.return
(end)
```

Table III. Similarity report for Table II.

Revision 2						
Function	SC	Number of calls	`Impact`	`TotalImpact`	`RangeDiff`	Runs
flushToDatabase()	1	50	0	0	600	3/3
generateReport()	0.58	50	496 B	24.8 KB	404	3/3
writeCache()	0	500	10 KB	5 MB	N/A	3/3

5.1. Data collection

To collect data on a function-level granularity, we must use a profiler or code instrumentation. In our implementation, we use Systemtap [11], a tool to simplify the gathering of information about a running Linux system. The difference between Systemtap and traditional profilers is that Systemtap allows dynamic instrumentation of both operating system (*system calls*) and application-level functions. Because of the ability of monitoring system calls, we can monitor applications that use libraries written in different languages. In addition, by instrumenting system calls, we can monitor data that are normally hidden from higher level languages such as the number of bytes written or allocated.

These advantages are illustrated by the following example. We want to monitor the number of bytes written by application-level functions of an application that uses libraries written in C and in Python so that we can find the functions that write the most during the execution of a test. Libraries written in C use different application-level functions for writing files than libraries written in Python. If we were to instrument these libraries on the application level, we would have to instrument all those functions. In addition, we would have to identify all writing functions in all libraries. However, after compilation or interpretation, all these functions use the same subset of system calls to actually write the file. Hence, if we could instrument those system calls and find out from which application-level function they were called, we can obtain the application-level information with much less effort.

By combining application-level and operating system-level data with Systemtap, we can obtain a detailed profile of the writing behavior of our application and any libraries that it uses. Systemtap allows dynamic instrumentation [11] by writing *probes* that can automatically instrument the entry or return of functions. Listing 1 shows the workflow through (a subset of) the available probe points in a Python function that writes to a file. Note that if we want to monitor other metrics such as network traffic, we must probe other system calls.[§]

```
probe begin{/* Print the CSV headers */}
probe python.function.entry{/* Add function name to the stack trace */}
probe syscall.open.return{/* Store the filehandler and filename of the
    opened file */}
probe syscall.write.return{/* Add the number of bytes written */}
probe python.function.return{/* Print the python stack trace and the number
    of bytes written */}
```

The subject system of our user study (refer to Section 6), Tribler, is written in Python. Therefore, we implemented a set of probes to monitor the number of bytes written per Python function. Listing 2 shows the description of this set of probes.[¶] By running these probes together with any Python application, we can monitor write I/O usage on a function-level granularity. Files may be written by other system calls such as *sys_writev* and *sys_sendfile*. Likewise, system calls other than *sys_open* may open files or obtain file descriptors. We have counted the number of calls to various system calls for writing files during an execution of the Django test suite described in Section 7. *sys_write* was called 292 000 000 times, *sys_writev* was called 236 times, and the other functions were not called. In addition, the total number of bytes written by those 236 calls to *sys_writev* was 39k. Therefore, for clarity of this text, we have only probed the *sys_write* and *sys_open* system calls during our studies.

While Systemtap natively supports C, C++, and Java, it does not include native support for probing Python programs. Therefore, we use a patched version of Python, which allows Systemtap to probe functions. This version of Python can be automatically installed using Gumby.

To monitor write actions, we count the number of bytes written per stack trace. To maintain a stack trace, for every Python function that we enter (*python.function.entry*), we add the function name to an array for that thread. This allows us to distinguish multiple threads of the application. Then, for all the writes performed during the execution of that function, we sum the total number of bytes written per file (*syscall.open.entry* and *syscall.write.entry*). We use the probe on the open system call only to map the file descriptors with a filename (per thread). We do this so that we can probe functions that

[§]Refer to http://man7.org/linux/man-pages/man2/syscalls.2.html for Linux system calls.
[¶]Refer to the Gumby source code for the exact implementation.

use the file descriptor. As a result, the open system call does not have to be made from within the function. After returning from the Python function (*python.function.return*), we output the number of bytes written per file for the function and the stack trace to that function in CSV format. As a result, we have a CSV file with the size and stack traces of all write actions during the test execution.

5.2. Data processing

After collecting the data, we import it into a SQLite[‖] database using R[**] and Python. From this database, we generate a report for each test execution (the *test execution report*) that shows the following:

1. The stack traces with the largest total number of bytes written.
2. The stack traces with the largest number of bytes written per call.
3. The filenames of the files to which the largest total number of bytes were written.

The test execution report helps with locating the write-intensive stack traces for this execution. In addition, when we have monitored all test executions for a revision, we generate a profile as described in the previous section. We use this profile as a basis to analyze test executions for the next revision.

6. FIELD USER STUDY: DISPERSY AND TRIBLER

We evaluate our approach in a field user study. The goal of our study is to determine whether I/O performance regressions can be found and optimizations can be verified using our approach. In particular, we focus on these research questions:

Eval RQ 1: Does our approach provide enough information to detect performance regressions?
Eval RQ 2: Does our approach provide enough information to guide the performance optimization process?
Eval RQ 3: Does our approach provide enough information to verify the effect of made performance optimizations?
Eval RQ 4: How does our approach work for test executions that are influenced by external factors?

In this section, we present the experimental setup of our field user study.

Field setting: The subject of our study is Tribler [7], a fully decentralized open-source BitTorrent client. Since its launch in 2006, Tribler was downloaded over a million times. Tribler is an academic prototype, developed by multiple generations of students, with approximately 100K lines of code. Tribler uses Dispersy [12] as a fully decentralized solution for synchronizing messages over the network. Tribler has been under development for 9 years.

One of the goals for the next version is to make it run better on older computers. Therefore, we must optimize the resource usage of Tribler. In the first part of our study, we analyze the unit test suite of Dispersy. In the second part, we analyze a 10-min idle run of Tribler, in which Tribler is started without performing any actions in the GUI. However, because of the P2P nature of Tribler, actions will be performed in the background as the client becomes a peer in the network after starting it.

Participant profile: The questionnaire was filled in by two participants. Participant I is a PhD student with 4 years of experience with Tribler. Participant II is a scientific programer with 5 years of experience with Tribler, in particular with the Dispersy component. Both participants describe their knowledge of Tribler and Dispersy as very good to excellent.

Experimental setup: Tribler and Dispersy are being maintained through GitHub.[††] We implemented a script in Gumby that does the following for each of the last *n* commits:

[‖]http://www.sqlite.org/
[**]http://www.r-project.org/
[††]http://www.github.com/tribler

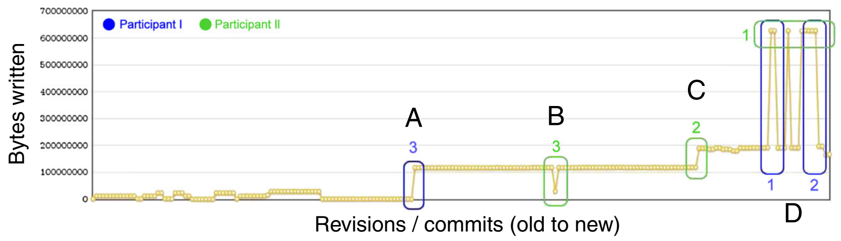

Figure 1. Average number of bytes written during an execution of the Dispersy unit test suite for each commit.

1. Execute the required test five times together with the Systemtap probes (this number was chosen based on the execution time of the tests—we have no statistical evidence that this is indeed an optimal value)
2. Load the monitored data into a SQLite database
3. Generate a test execution report for each test execution as explained in Section 5.2
4. Compare the output of each run with the previous revision and add this result to the activity matrix *m*
5. Calculate SC for every row in *m*
6. Generate a similarity report from the activity matrix as displayed in Table III
7. Generate a profile to compare with the next revision

After all commits have been analyzed, the data are summarized in an *overview* report. The overview report shows a graph (e.g., Figure 1) of the average number of total bytes written for the test executions of a revision/commit and allows the user to drill down to the reports generated in steps 3 and 6, that is, each data point in the graph acts as a link to the similarity report for that commit. Each similarity report contains links to the test execution reports for that commit. In addition, we added a link to the GitHub diff log for each commit so that the participants could easily inspect the code changes made in that commit.

In the Dispersy case study, we will analyze the unit test suite of Dispersy for the last 200 revisions. In the Tribler case study, we will analyze a 10-min idle run of Tribler for the last 100 revisions. Tribler needs some time to shutdown. If for some reason, Tribler does not shutdown by itself, the instance is killed after 15 min using a process guard. Note that these numbers were chosen based on the execution time of the tests. We have no statistical evidence that these are indeed optimal values.

All experiments were conducted on a dual Intel Xeon CPU 2.40 GHz with 12 cores and 8 GB of memory, running Debian with a custom[‡‡] complied kernel 3.12.

Questionnaire: To evaluate our approach, we asked two developers from the Tribler team to rate the quality and usefulness of the reports. We presented them with the reports for the Dispersy and Tribler case study and asked them to do the following:

1. To select the three most interesting areas (5–10 data points) on the graphs and rate them 1 (first to investigate) to 3 (third to investigate)
2. To mark with 1–3 the order of the points that they would investigate for each area

Then, for each area/phenomenon and each selected data point, we asked them to answer the following:

1. Which position shows the stack trace that you would investigate first/second/third, based on the report?
2. Does this lead to an explanation of the phenomenon, and if so, which one?
3. If not, please drill down to the separate test execution reports. Do these reports help to explain the phenomenon?

Finally, we asked them general questions about the reports concerning the usability and whether they expect to find new information about Tribler and Dispersy using this approach. In the next section, we present the results of our study. The reports can be found online [13].

[‡‡]Systemtap requires the uprobes kernel module for user-space probing, which is not available by default in all kernels. For more information, refer to https://sourceware.org/systemtap/SystemTap_Biginners_Guide/userspace-probing.html.

Table IV. Overview of Dispersy evaluation results.

Phenomenon	Participant	Number of ranking	Helpful?
A	I	1	Yes
B	II	84	No
		Test execution reports	No
C	II	1	Partly
		18	Yes
D	I (area 1)	1	Yes
	I (area 2)	1	Partly
	II	1	Yes

6.1. *Case study I: dispersy unit test suite*

Every run of the Dispersy unit test suite took approximately 10 min. As a result, the experiment for 200 revisions and five iterations ran for 10 000 min, approximately 7 days. While we did not do this, it is possible to run these executions in parallel, reducing the total running time. The unit test suite had a coverage[§§] of 73%. During a test suite execution, an average of 130 different stack traces doing writes was monitored. The average length of a stack trace was 9. Figure 1 contains the graph generated during the Dispersy study. In the graph, we highlighted the areas marked by the participants (including their rankings for the most interesting ones). Both participants selected phenomenon D as the most interesting to investigate because of the increased writes of over 400 MB. Participant I considered the peaks as separate phenomena, while participant II considered them as one event. Furthermore, participant II expected that the cause of phenomenon A was the addition of test cases that resulted in more I/O; hence, he selected different phenomena to investigate. Next, we discuss each phenomenon and the way that the participants investigated them. Table IV gives an overview of which ranked position the participants analyzed and whether the information provided was useful.

Phenomenon A: The increase was caused by a bug fix. Before this bug fix, data were not committed to the database.

Participant's analysis: Participant I indicated that our ranking correctly showed that the database commit function started doing I/O or was called since the previous commit.

Phenomenon B: The drop in writes is because of the order in which the Git commits were traversed.

Git allows branching of code. In this case, the branch was created just before phenomenon A and merged back into the main branch in phenomenon B. In Git, a pull request can contain multiple subcommits. When requesting the Git log, Git returns a list of all commits (including subcommits) in topological order. This means that every merge request is preceded directly by its subcommits in the log. Hence, these commits were traversed by us first. Figure 2 shows an example for the traversal order of a number of commits.

Likewise, the drop during phenomenon B was caused by testing 'old' code, which leads to a confusing report. For the Tribler and Dispersy projects, short-living feature branches are used. As a result, we can avoid the traversal order problem by testing only merge requests on the main branch, without subcommits. However, this would also make the analysis of the cause more difficult as the number of changes to the code is larger when subcommits are combined ('squashing' in Git terminology). In other projects, which use large long-living branches, it may be better to follow all commits within the branch. Hence, the traversal should be selected depending on the way branches are used in the project. In future work, we will investigate in more projects how the traversal order affects the analysis results.

Participant's analysis: Participant II was not able to explain this effect from the report. However, after explaining this effect, the phenomenon was clear to him.

Phenomenon C: In the updated code, a different test class was used that logged more info.

Participant's analysis: Participant II indicated that he inspected the similarity reports for the highest and the lowest point of the phenomenon. From the report for the highest point, he suspected the

[§§]Calculated with http://nedbatchelder.com/code/coverage/

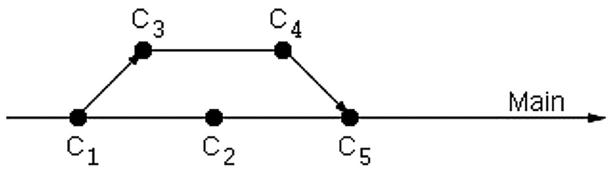

Figure 2. Order of traversal of commits in Git log (C_1 to C_5).

number 1 ranked stack trace caused the phenomenon. However, as he was not convinced yet, he used the report for the lowest point to verify his suspicions, in which this stack trace was ranked number 18. From the combination of the reports, he concluded the number of calls changed from 270 to 400, causing the phenomenon. After inspecting the code changes using the GitHub diff page, he concluded that the different test class was the cause for the increase in the number of calls.

Because the participant was not convinced by the number 1 ranked stack trace by itself, we marked this stack trace as 'partly useful' in Table IV. Following the advice from participant II, the reports were extended with the `CallsDiff` metric after the user study. This metric shows the difference in the number of calls to each stack trace, compared with the previous revision.

Phenomenon D: A new test case creates 10k messages and does a single commit for every one of these messages, introducing an additional 435 MB of writes.

Participant's analysis: Participant I marked this phenomenon as two separate events, for the same reason as explained for phenomenon B. Both participants were able to explain and fix the issue based on the highest ranked stack trace in the report. This was the trace in which a commit is made to the database, which had an SC of 0 and a `TotalImpact` of 435 MB. As the number of calls was 10k, the participants fixed the issue by grouping the commits. The fix was verified using our approach. From the graph, we could see that the total writes decreased from 635 to 200 MB. From the similarity report, we found that the number of calls to the stack trace decreased from 10k to 8.

6.2. *Case study II: Tribler idle run*

During a 10-min idle run, an average of 130 different stack traces doing writes was monitored. The average length of a stack trace was 14. Figure 3 contains the graph generated during the Tribler case study. We have marked the areas selected by the participants. It is obvious that this graph is less stable than the Dispersy graph. The reason for this is that the behavior during the idle run (i.e., just starting the application) is influenced by external factors in Tribler. Because of its decentralized nature, an idle client is still facilitating searches and synchronizations in the background. As a result, the resource usage is influenced by factors such as the number of peers in the network. Despite this, the participants both selected phenomena C and D as interesting. Participant II explained later that the difference in the choice for A and B was because he preferred investigating more recent phenomena, as their cause is more likely to still exist in the current code. In the remainder of this section, we discuss the phenomena and the participants' evaluations. Table V summarizes the results for the Tribler study.

Phenomenon A: During two out of five test executions, Tribler crashed for this commit. Hence, fewer messages were received, resulting in a lower average of bytes sent. The actual explanation for this crash cannot be retrieved from these reports but should be retrieved from the application error logs.

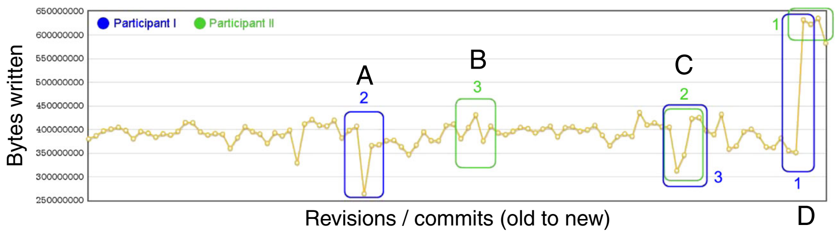

Figure 3. Average number of bytes written during a 10-min Tribler idle run for each commit.

Table V. Overview of Tribler evaluation results.

Phenomenon	Participant	Number of ranking	Helpful?
A	I	45	Partly
B	II	—	No
C	I	1	No
	I	65	Partly
	II	—	No
D	I	1	Partly
	II	24, 26, 27	No
	II	17, 31, 2	Partly

Participant's analysis: From the reports, participant I was able to detect that fewer messages were received, but he was not able to detect the actual cause for this. Therefore, he granted the behavior to noise because of external factors. Because the participant was able to analyze the symptom correctly from our reports but could not make the correct diagnosis, we classify this report as 'partly helpful' in Table V.

Phenomenon B: No significant changes were found; the variation was because of external factors.

Participant's analysis: Participant II correctly diagnosed this as noise because of external factors.

Phenomenon C: There was no clear explanation for the drop in resource usage. It was probably because of less active users in the network during the test execution.

Participant's analysis: Both participants concluded that fewer messages were received and that the phenomenon did not require further investigation.

Phenomenon D: The reason for the large increase in writes is that the committed code made was part of Tribler crash. As a result, the idle run had to be killed after 15 min by the process guard. This allowed the part of Tribler that still was running to collect data longer than during the other runs, with the high peak in the graph as the result.

Participant's analysis: Both participants correctly indicated that more messages were received, and they could both identify the function that caused the large number of writes. They did not directly indicate the partial crash as the cause. Both participants advised to include the following: (i) the actual duration of the execution and (ii) a link to the application logs in the report, in order to be able to diagnose such cases better in the future.

In addition, the participants agreed that the function causing the large number of writes used too much resource. This resulted in a performance optimization, which was validated using our approach. From the reports of the validation, we could see that the total number of written bytes decreased by 340 MB after the fix, and from the similarity reports, we could see that the stack trace disappeared from the report. This means that the function stopped doing write I/O.

6.3. Evaluation results

From our evaluation, we obtain an indication that our approach is useful for finding performance regressions. Especially in the case of a test that is repeatable, such as the Dispersy test suite, our approach leads to detection of performance regressions, which themselves point in directions for optimization. For test suites that are influenced by external factors, such as the Tribler idle run, our analysis results require deeper investigation and may show more phenomena that are either difficult to explain using our reports or simply are not performance regressions.

Even so, the participants were able to correctly analyze and diagnose three out of four phenomena in the Dispersy report. In addition, they were able to partly analyze two out of four phenomena in the Tribler report. The participants indicated that with little more information, they would have been able to correctly diagnose all phenomena. These results are summarized in Table VI. Together with the participants, we concluded that the reports miss the following information:

1. The `CallsDiff` metric, which displays the difference in the number of calls to a function via the path showed in the stack trace between two commits
2. A link to the application log so that the user of the report can check for the exit code and whether any exceptions occurred during the test execution

Table VI. Summary of field user study results

Dispersy	Participant	Correct?	Tribler	Participant	Correct?
A	I	Yes	A	I	Partly
	II	—		II	—
B	I	—	B	I	—
	II	No		II	No
C	I	—	C	I	Partly
	II	Yes		II	No
D	I	Yes	D	I	Partly
	II	Yes		II	Partly

3. The total duration of the test execution
4. An explanation of (or solution to) the 'Git log order' effect, explained in Section 6.1

After the user study, one phenomenon (Dispersy phenomenon D) was optimized based on our reports. In addition, one regression (Tribler phenomenon D) was fixed. While this was not directly a performance optimization, investigation was triggered by our reports. Both the optimization and the fix could be validated using our approach after they were made. During the case study, a phenomenon was also correctly explained to be a validation of a performance bug fix. Finally, according to the participants, four out of the five phenomena, which did not represent a performance regression, were easy to diagnose.

In Tables IV and V, we see that in the Dispersy study, the problem was indicated by the top ranked stack trace in most cases. In the Tribler study, this is not the case, but the lower ranked stack traces were selected because of their high negative impact. If we would rank the traces by the SC and absolute value of `TotalImpact` (instead of exact value), the traces would have had a top three rank as well. Hence, we can conclude that the ranking given by our approach is useful after a small adjustment. An observation that we made was that the participants all used the `TotalImpact` as a guideline for indicating whether the change in behavior of a stack trace was significant enough to investigate further. After this, they checked the SC to see in how many test executions the behavior was different. This indicates that the ranking should indeed be made based upon a combination of these two metrics and not by the SC or `TotalImpact` alone.

7. FEASIBILITY STUDY: DJANGO

To ensure the external validity of our approach and to show that our approach works for projects developed outside our laboratory, we have applied it to Django.[¶¶] Django is a web framework that allows for rapid development and clean design of web applications. Django is used by large websites such as Instagram and Pinterest.[***] Django is written in Python and is currently the most popular Python project on Github.[†††] The Django Git repository[‡‡‡] contains over 15k commits since 2005. For this feasibility study, we have analyzed the write behavior of the test suite of Django of the following revisions:

1. All (20) revisions of which the commit message contains the word 'performance'
2. The last 100 'merge' commits
3. Thirty-three (sub)commits causing the phenomenon found during the 'merge' commit traversal

In contrast to the study described in Section 6, we did not conduct a user study but have conducted the analysis of Django ourselves. The setup for this feasibility study was equal to the setup described in Section 6, with the addition that we show a warning in the similarity report if a revision contained one or more crashing test suite executions. The reports can be found online [13].

[¶¶]https://www.djangoproject.com/
[***]http://www.instagram.com/ and http://www.pinterest.com
[†††]http://pythonhackers.com/open-source/
[‡‡‡]http://www.github.com/django/django

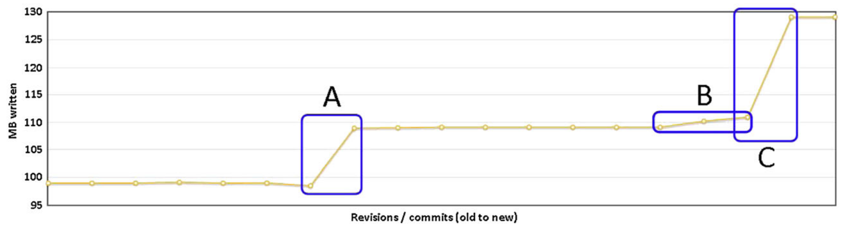

Figure 4. Average number of MB written during execution of the Django test suite (‘performance’ commits).

The unit test suite had a coverage of 91%. During a test suite execution, an average of 426 different stack traces doing writes was monitored. The average length of a stack trace was 13.

Every run of the Django unit test suite took approximately 11 min. As a result, the experiment for 100 revisions and five iterations ran for 5500 min, approximately 3.5 days. Configuring Gumby to run the Django experiment took 2 h.

7.1. 20 ‘performance’ commits

Figure 4 shows the overview report of the revisions of which the commit message contains the word ‘performance’. In total, there were 45 performance commits, but we were only able to run the latest 20, mostly because of dependence problems as the older commits were more than 2 years old. Therefore, we only graphed the latest 20 performance commits.

Figure 4 shows three regressions. Because the commits were made over a period of 2 years, spanning 4346 commits in total, these regressions are likely because of the growth of the application and of the unit test suite. In the next two sections, we will analyze the largest regression, phenomenon C, in more detail. We have analyzed phenomenon C only because of the large number of test executions required to analyze all subcommits. In a production situation, this would not be an issue, as it is possible to run the tests in parallel with the development.

7.2. 100 merge commits

Figure 5 shows the overview report of the last 100 ‘merge’ commits made to the Django repository. We analyzed only the merge commits to avoid the Git traversal order problem, as seen in the Dispersy case study. Note that this commit range was selected to include the commits of phenomenon C in Figure 4.

Phenomena C.1 and C.2 were caused by crashing revisions, and we have marked them manually as such in the similarity report. In the next section, we will investigate phenomenon C.3 in more detail. Note that this is the same phenomenon as phenomenon C in Figure 4.

7.3. 33 subcommits causing the phenomenon of Section 7.2

Because Figure 5 contains merge commits only, we must analyze all commits made between the two commits of phenomenon C.3 to precisely pinpoint the code causing the increase in writes.

Figure 6 shows the overview report of the 33 commits made between the two commits surrounding phenomenon C.3 of Figure 5. From Figure 6, in the results of Section 6 and the Git commit messages, we can deduce that phenomenon C.3.2 is caused by the Git traversal order: The last commit that we analyzed was the commit in which the commit causing the drop of C.3.2 was merged into the main branch. Hence, we could verify that the commit causing the drop was using code from the main branch that was written before phenomenon C.3.1. After this observation, it is obvious that the increase in writes was caused by the code changes in the revision on the high side of phenomenon C.3.1. After closer inspection of the changes made in this revision,[§§§] we found that this code implements cache-like functionality in Django. In order to allow a website to use an older revision of files, such as stylesheets, Django adds the MD5 hash of the content to the filename. To avoid the

[§§§]https://github.com/django/django/commit/8efd20f96d2045cf08baded98e18d241e4c6122d

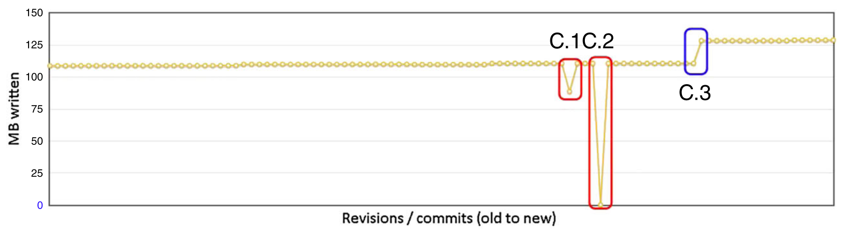

Figure 5. Average number of MB written during execution of the Django test suite (last 100 merge commits).

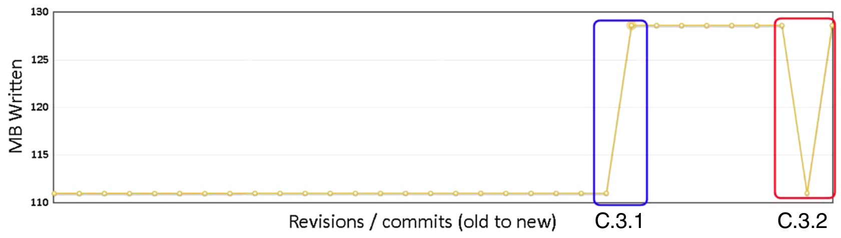

Figure 6. Average number of MB written during execution of the Django test suite (33 subcommits).

performance burden caused by creating the MD5 hash every time the file is accessed, it can be preprocessed from the administration panel. This preprocessing is performed by calling the `collectstatic` management command, which was already available in previous revisions to perform other similar functionality. After adding the MD5 hash code to the code executed by `collectstatic`, it generates more writes because the file with the hash in it is copied. Our similarity report correctly indicated that the `collectstatic` command was the number 1 suspect cause of the increased writes. Because the increase in writes was intended, we can conclude that this is an 'acceptable' performance regression; hence, it does not require a fix.

8. DISCUSSION

8.1. The evaluation research questions revisited

Does our approach provide enough information to detect performance regressions? From our evaluation, we get a strong indication that our reports provide, after adding the information explained in Section 6.3, enough information for detecting I/O performance regressions. In our user study, one out of two detected regressions was diagnosed correctly by the participants. The participants were able to partly analyze and diagnose the second one. In addition, we were able to detect three performance regressions in the Django study, of which we analyzed one in detail.

Does our approach provide enough information to guide the performance optimization process? Our evaluation showed that our approach is capable of detecting I/O performance regressions in the Dispersy, Tribler, and Django projects. These regressions can act as a guide during the search for possible performance optimizations. Our user study alone resulted in an optimization (Dispersy phenomenon D) and a fix (Tribler phenomenon D) that have immediately been carried through in the respective projects. Hence, we obtain an indication that our approach can guide the performance optimization process. In future work, we will conduct more case studies to get stronger evidence for this claim.

Does our approach provide enough information to verify the effect of made performance optimizations? Our approach provides enough information to validate the optimization and fix made after the user study. In addition, the participants were able to validate a performance fix made in the history of Dispersy. The participants indicated that the optimizations would have been easier to validate if the difference in number of calls for each stack trace was shown in the reports; hence, these have been added to our approach.

How does our approach work for test executions that are influenced by external factors? From our Tribler case study, we get the indication that our approach should be adapted before it can properly deal with influence from external factors, as the participants were able to partly explain two out of four performance phenomena only. However, our case study does show that the reports provided useful and new information to the participants. Hence, we will continue to pursue our effort, and in future work, we will do research on how we can minimize the effect of external factors.

8.2. Scalability and limitations

For the moment, the overhead of our approach is considerable, mostly because of our inefficient implementation of the monitoring probes for Systemtap. The execution time of the Dispersy test suite increased from 320 s to approximately 550 s when the monitor was enabled. Likewise, the running time of the Django test suite increased from 325 s to approximately 660 s with the monitor enabled. While we expect that this does not affect relatively stable metrics such as I/O, we acknowledge that this must be improved to be able to monitor timing-sensitive metrics, such as execution time. In addition, to the best of our knowledge, Systemtap is the only available option for monitoring Python code with such granularity. Therefore, we will focus on minimizing the overhead of our approach in future work.

In this paper, we focused on write I/O. We set up our tooling infrastructure such that the monitoring component can easily be exchanged for another component that is able to monitor different metrics. Hence, by changing the monitoring component, our approach can analyze other performance metrics. In addition, we will investigate how we can rank stack traces on a combination of these metrics, rather than on one metric only. This would help in making a trade-off between the various performance metrics while optimizing.

A limitation of our approach is the fact that we require stack traces to be monitored. Some existing profilers (such as the default Python profilers[§§§]) do not offer this functionality. As a result, we must resort to tools such as Systemtap to collect the data. While Systemtap is a powerful tool, it is more difficult to use and install than most profilers. This limits the ease with which our approach is adopted.

We have implemented our approach for Python programs only. However, because Systemtap natively supports C, C++, and Java, we expect that our approach will work for programs implemented in (a combination of) those languages as well. In future work, we will verify that this is indeed the case.

Another limitation is that we compare a version with its predecessor only. In future work, we will investigate if comparing with more versions can lead to new insights, such as the detection of performance degradation over longer periods.

In our approach, we do not deal with errors that occurred during the test executions. When no profile could be generated for a revision, we simply compare with the last revision that has a profile. In future work, we will investigate how we can inform the user about errors better, for example, by using information from the application logs in our reports.

8.3. Threats to validity

With regard to external validity, we can report that we have performed our field study on Tribler and Dispersy, a set of applications that has been under development for 9 years and is downloaded over a million times. In order to make sure that our approach generalizes to nonacademic-oriented software systems, we have performed a feasibility study on Django, the most popular Python project on Github with 538 contributors and more than 10 000 commits. We acknowledge that future work should concentrate on gathering more evidence for the usefulness of our approach through additional case studies.

The field study on Tribler and Dispersy was conducted with two users only. While the user study was carried out with developers who have considerable experience with the application, we acknowledge that in the additional case studies that we plan to perform, we should involve more developers.

[§§§]https://docs.python.org/2/library/profile.html

Concerning the internal validity of our approach, we acknowledge that using the range of monitored values in the profiles is not a statistically sound method. However, because of the low number of test executions, we feel that using a value such as the standard deviation (*std*) does not add to the reliability of the profiles. To verify this, we have calculated the *mean* and *std* for Django and Dispersy executions. We found that for the Django case, the *std* was 0 for all cases, which means that there was no difference in writes between the test executions for one revision. As a result, we could have used a lower number of executions. For Dispersy, we manually compared the ranges created by our approach, $mean \pm std$ and $mean \pm 2{*}std$. For these revisions, the range created by $mean \pm std$ was, on average, slightly smaller than the range created by our approach, while $mean \pm 2{*}std$ resulted in slightly larger ranges. Hence, we feel that for five test executions, it does not really matter which approach is used. In future work, we will do more analysis on the optimal number of test executions and the statistical approach to use.

We acknowledge that the performance of an application is heavily influenced by the version of the (external) libraries and kernel it uses. To keep this threat as small as possible, we have executed our case studies from within a virtual environment. Gumby contains a build script for this environment, allowing the case studies to be repeated in an environment that always has the same dependence installed.

8.4. Lessons learned

In our studies, we have used the unit test suite to do performance regression tests. The advantage of using unit tests is that they are usually readily available, while performance tests are not. This lowers the bar for developers to do performance tests. In addition, unit testing can be performed early in the development cycle, allowing potential performance issues to be caught early, possibly making their solution cheaper to implement. The disadvantage of using the unit test suite is that it was not designed to stress the performance of the application. In addition, it is possible that the analysis leads to optimizations of the unit test suite only. However, we believe that a unit test suite should contain representative behavior of the application. Therefore, we expect that performance regressions occurring in the test suite are likely to occur in the application itself. In future work, we will investigate how our choice of using the unit test suite affects the analysis results of our approach.

During our studies, we learned that the filename of the file written to was not always correctly captured by our approach. This was because of the fact that we assumed that only one file descriptor is used within a function. For the Tribler project, this was often the case, because of the style of programing. In addition, we learned from a discussion with the developers that they usually had a good indication of what was causing the writes by just knowing the stack trace doing the writes. While we feel that this wrong assumption did not affect the results of our case studies, we will fix this issue in future work by taking our monitoring granularity one step further and monitoring writes per file descriptor per function, instead of just per function.[¶¶¶]

Although we did not encounter this during our studies, special care must be taken in the case of refactoring. Because refactored code paths are considered new, they will receive a high ranking in the similarity report if they perform writes. If new code, which performs writes, is introduced as well in the same revision, analysis results can be confusing. This can be avoided by separately committing refactoring changes and new code (i.e., *self-contained* commits [14]).

The final lesson that we learned is that it is useful to do performance regression testing in an automated fashion using a continuous integration environment such as Jenkins.[‖‖‖] This is because automation does as follows: (i) allows for quick feedback, which, in turn, will help in finding early solutions to performance issues, and (ii) will help in lowering the bar for software developers to incorporate regression testing into the development cycle.

[¶¶¶]Note that this was implemented in the latest version of Gumby, which was developed after performing the experiments.

[‖‖‖]http://jenkins-ci.org/

9. RELATED WORK

Comparison of execution profiles and the detection of performance regressions have received surprisingly little attention in research. Savari [15] has proposed a method that works for frequency-based profiling methods. Our approach works for any type of metric on a function-level granularity.

The widely used profiler OProfile**** implements a technique known as differential profiles, which expresses differences between profiles in percentage. The problem with this technique is that it reports high percentages for the difference of small values. For example, the difference between 1 and 2 KB is 100%, while the absolute difference is relatively small. As a result, it is unknown whether this difference is significant, without incorporating the number of calls to the function. Our approach calculates the impact of the code change instead, which does not have this problem.

Bergel *et al.* [16] have proposed a profiler for Pharo that compares profiles using visualization. In their visualization, the size of an element describes the execution time and number of calls. Alcocer [17] extends Bergel's approach by proposing a method for reducing the generated call graph. These visualizations require human interpretation, which is difficult when the compared profiles are very different [16]. Our approach provides a textual ranking, which we expect to be easier to interpret. However, we believe that the work of Bergel *et al.* and Alcocer and our approach can be supplemental to each other, and we will investigate this in future work.

Jiang *et al.* [18] analyze readily available execution log files to see if the results of a new load test deviate from previous ones. The advantage of this approach is that it does not introduce extra monitoring overhead. However, this also limits the granularity with which regression analysis can be performed. This is also demonstrated by the granularity of their case studies: In three conducted case studies, they analyze system and application-wide tasks such as finding the optimal database management system configuration. Our approach does not have such a limitation. However, this comes at the cost of increased overhead.

Nguyen *et al.* [19] propose an approach for detecting performance regressions using statistical process control techniques. Nguyen *et al.* use control charts to decide whether a monitored value is outside an accepted range. The violation ratio defines the relative number of times that a value is outside this range. Control charts and the violation ratio are similar to our profile approach. The approach of Nguyen is more statistically sound than our approach; however, we expect that this is not necessarily an improvement when using a small number of test executions. The main difference in the approach used by Nguyen and our approach is the granularity. Their approach identifies performance regressions in system-level metrics, while our approach identifies regressions on the function level, making analysis of the regression easier. In future work, we will investigate how our approach and Nguyen's approach can complement each other.

10. CONCLUSION

In this paper, we proposed a technique for detecting and analyzing I/O performance regressions. By comparing execution profiles of two software versions, we report on the functions of which the performance profile changed the most. This report can be used to find regressions or to validate performance optimizations. In this paper, we focused on analyzing write I/O regressions, which is a relatively stable metric. In future work, we will investigate how our approach performs when applied to less stable, timing-sensitive metrics, such as execution time and CPU usage.

In a field user study, we showed that our approach provides adequate information to detect write I/O regressions and guides the performance optimization process. In fact, our field user study resulted in one optimization and one fix made to our subject system. In addition, we show that our approach was able to detect three performance regressions in one of the most popular Python projects, namely Django. To summarize, we make the following contributions:

1. An approach for the detection and analysis of I/O performance regressions

****http://oprofile.sourceforge.net/

2. An open-source implementation of this approach, called SPECTRAPERF
3. A user study in which we show that our approach guides the performance optimization process
4. A feasibility study in which we show that our approach works on applications developed outside our laboratory

Revisiting our research questions,

RQ 1: How can we monitor performance data and generate a comparable profile out of these data? We have proposed an approach using Systemtap to monitor data, and we have showed how to generate a comparable profile from these data.

RQ 2: How can we compare the generated performance profiles? We have presented our approach for comparing performance profiles and provide a ranking of the stack traces that were most likely to have changed behavior. This ranking is made based on the *similarity coefficient* compared with the previous performance profile and the *total impact* of a source code change on performance. In our user study, we showed that the ranking was useful in three out of eight cases and partly useful in three out of eight cases and helped the participants find one optimization and one fix. In addition, we were able to detect three performance regressions in Django, one of the most popular Python projects.

RQ 3: How can we analyze and report on the differences between profiles? We have showed how we report on the data, and we have evaluated this reporting technique in a field user study. During this study, we analyzed the performance history of the open-source P2P client Tribler and one of its components, Dispersy.

The field user study resulted in one optimization and one fix, which were also validated using our approach. During the user study, we found that our approach works well for repeatable tests, such as a unit test suite, as the participants were able to explain three out of four performance phenomena encountered during such a test using our approach. We also received an indication that it needs adaptation for a test that was influenced by external factors, as the participants were able to explain two out of four performance phenomena partly only.

Main RQ: How can we guide the performance optimization process by doing performance regression tests? We have showed that our approach for performance analysis can guide the performance optimization process by detecting I/O performance regressions. The results of our field user study alone resulted in one optimization for Dispersy and a fix for Tribler.

In future work, we will focus on doing more case studies and on extending our approach to monitor different performance metrics such as memory and CPU usage. Additionally, we will investigate how we can report on trade-offs between these metrics.

REFERENCES

1. Rothermel G, Harrold MJ. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on* 1996; **22**(8):529–551.
2. Chung L, Nixon B, Yu E, Mylopoulos J. *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishers: Boston/Dordrecht/London, 2000.
3. Jain R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons: New York, 1991.
4. Reiss SP. Visualizing the java heap to detect memory problems. Int'l Workshop Visualizing Software for Understanding and Analysis (VISSOFT), IEEE, 2009; 73–80.
5. Woodside M, Franks G, Petriu D. The future of software performance engineering. Future of Softw. *Engineering (FOSE), IEEE* 171–187. doi:10.1109/FOSE.2007.32
6. Bezemer CP, Zaidman A. Performance optimization of deployed software-as-a-service applications. *Journal of Systems and Software* 2014; **87**(0):87–103.
7. Pouwelse JA, Garbacki P, Wang J, Bakker A, Yang J, Iosup A, Epema DH, Reinders M, Van Steen MR, Sips HJ. Tribler: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience* 2008; **20**(2):127–138.
8. Larres J, Potanin A, Hirose Y. A study of performance variations in the mozilla firefox web browser. Proc. Australasian Comp. Science Conference (ACSC), Australian Computer Society, Inc., 2013; 3–12.
9. Abreu R, Zoeteweij P, Van Gemund AJ. On the accuracy of spectrum-based fault localization. Testing: Academic and Industrial Conf. Practice and Research Techniques-MUTATION (TAICPART-MUTATION), IEEE, 2007; 89–98.

10. Abreu R, Zoeteweij P, van Gemund A. An evaluation of similarity coefficients for software fault localization. Pacific Rim International Symposium on Dependable Computing (PRDC), 2006; 39–46.
11. Prasad V, Cohen W, Eigler F, Hunt M, Keniston J, Chen B. Locating system problems using dynamic instrumentation. Proc. Ottawa Linux Symposium, 2005; 49–64.
12. Zeilemaker N, Schoon B, Pouwelse J. Dispersy bundle synchronization. Technical Report PDS-2013-002, TU Delft 2013.
13. Bezemer CP. Performance regression reports 2014. URL 10.6084/m9.figshare.974535, figshare.
14. Mulder F, Zaidman A. Identifying cross-cutting concerns using software repository mining. Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL, ACM: New York, NY, USA, 2010; 23–32.
15. Savari SA, Young C. Comparing and combining profiles. *Journal of Instruction-Level Parallelism* 2000; **2**:1–20.
16. Bergel A, Bañados F, Robbes R, Binder W. Execution profiling blueprints. *Software Practice Experience* 2012; **42**(9):1165–1192.
17. Alcocer JPS. Tracking down software changes responsible for performance loss. *Proceeding Int'l Workshop on Smalltalk Technologies (IWST), ACM* 2012; 3:1–3:7.
18. Jiang ZM, Hassan A, Hamann G, Flora P. Automated performance analysis of load tests. Prof. Int'l Conf. Softw. Maintenance (ICSM), IEEE 125–134. doi:10.1109/ICSM.2009.5306331
19. Nguyen TH, Adams B, Jiang ZM, Hassan AE, Nasser M, Flora P. Automated detection of performance regressions using statistical process control techniques. Proc. ACM/SPEC Int'l Conf. on Performance Engineering (ICPE), 2012; 299–310.