Delft University of Technology
Software Engineering Research Group
Technical Report Series

Detecting and Analyzing Performance Regressions Using a Spectrum-Based Approach

Cor-Paul Bezemer, Elric Milon, Andy Zaidman, Johan Pouwelse

Report TUD-SERG-2013-020

TUD-SERG-2013-020

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Detecting and Analyzing Performance Regressions Using a Spectrum-Based Approach

Cor-Paul Bezemer, Elric Milon, Andy Zaidman, Johan Pouwelse
Delft University of Technology, the Netherlands
{c.bezemer, e.milonbeltran, a.e.zaidman, j.a.pouwelse}@tudelft.nl

***Abstract*—Regression testing can be done by re-executing a test suite on different software versions and comparing the outcome. For functional testing, this is straightforward, as the outcome of such tests is either pass (correct behaviour) or fail (incorrect behaviour). For non-functional 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 performance regressions using a spectrum-based technique. 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. 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. 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.

I. 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 done by re-executing a test suite on different software versions and comparing the test suite outcome. For functional testing, this is straightforward, as the functionality of a program is either correct or incorrect. Hence, the outcome of such tests is either pass or fail. For non-functional testing, this is more challenging, as correct and incorrect are not clearly defined concepts for these types of testing [2].

An example of non-functional testing is performance testing. Two possible reasons for performance testing are:

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

SLA limits are often specified as hard thresholds for execution/response time, i.e., 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 comparison with the previous software version instead, as the goal is to make a task perform as fast or efficient as possible. 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 done on various metrics. Execution time, which is the most well-known, can be analyzed using traditional profilers. Other examples of metrics which can be optimized are the amount of I/O, memory usage and CPU usage. These metrics are difficult to analyze for software written in higher-level languages, such as Python, due to 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 which helps performance experts understand how the performance, including the metrics mentioned above, changes over the different versions of their software. 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. We achieve this by monitoring the execution of a specific test by two versions of an application and comparing the results using an approach based on *spectrum-based analysis* [6]. The result of our approach is a report which helps a performance expert to:

1) Understand the impact on performance of the changes made to the software on a function-level granularity
2) Identify potential performance 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 minute run of Tribler in the wild.

The outline of this paper is as follows. In the next section, we first give two motivational examples for our approach. In Section III, we present our problem statement. In Section IV, we explain spectrum-based fault localization, a technique which forms the basis for our approach. In Section V, we present our approach for spectrum-based performance analysis. We present the implementation of our approach, called SPECTRAPERF, in Section VI. In Section VII and VIII, we present the setup and results of our user study. We discuss these results and the limitations of our approach in Section IX. In Section X, we discuss related work. We conclude our work in Section XI.

II. Motivational Examples

In this section, we give two real-world motivational examples for the approach presented in this paper.

Monitoring I/O: In a database system, some queries require the creation of a temporary table[1]. The creation of such a file is often done silently by the database system itself, but is intensive in terms of I/O usage. 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 behaviour 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 due to the same reasons. Using an approach which can automate this 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.

Memory Usage: In many applications, custom caching mechanisms are used. Understanding the impact of these mechanisms on memory and disk I/O is often difficult. By being able to compare versions of software with implementations of different caching mechanisms, we can improve our understanding of their behaviour better. Through this better understanding, we can select, evaluate and optimize the most suitable caching mechanism for an application.

III. 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 1) be warned of undesired negative effects or 2) validate the positive effect of a performance bug fix. To give this feedback, we must do the following:

1) Define which metrics we want to analyze and combine this set of metrics into a *performance profile*, which describes the performance behaviour 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:

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 this data?*

Depending on which metric we want to analyze, we must find a suitable monitor (or profiler) to monitor 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 programming languages, making it more difficult to get fine-grained information about, for example, I/O.

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

A challenge is formed by the fact that monitoring the same test twice may result in slightly different performance profiles, due to variations in, for example, data contents and current memory usage [8]. 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 based on spectrum-based fault localization (see Section V). In this study, we focus on detecting and analyzing performance regression caused by write I/O. We expect that our approach can easily be adapted to work for other performance metrics, which we will verify in future work.

IV. Spectrum-Based Fault Localization (SFL)

Spectrum-based fault localization (SFL) is a technique that automatically infers a diagnosis from symptoms [9]. The *diagnosis* is a ranking of faulty components (block, source code line, etc.) in a system, with the most likely faulty one ranked on top. To make this ranking, observations are made during test execution. These observations express the involvement of components during that specific test case in *block-hit spectra* (hence the name of the technique). These spectra contain a binary value for each component, which represents whether it was executed during that test case. Together with the outcome of a test case (*pass/fail*), these observations form so-called *symptoms*. The outcome of all test cases (*0 = pass, 1 = fail*) is represented by the *output vector*.

All observations combined with the output vector form the activity matrix, which gives an overview of how component involvement is spread over the execution of a test suite. For every row in the activity matrix, the *similarity coefficient* of that row and the output vector is calculated. The idea behind this is that the row with the highest similarity coefficient indicates the component most likely to be faulty, as this component was executed during most of the failed test cases.

As the similarity coefficient, any similarity coefficient can be used, but Ochiai was proven to give the best results [10], hence we will use it throughout this study. This technique mimics how a human would diagnose an error by looking which parts of the system were active during the failed tests. The Ochiai similarity coefficient (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 the number of items in both vectors, b the number of items in v_1 that are not in v_2 and c the number of items

TABLE I
ILLUSTRATION OF SFL [9]

Component	Character counter	t_1	t_2	t_3	t_4	t_5	t_6	SC
	def count(string)	[Activity Matrix]						
C_0	let = dig = other = 0	1	1	1	1	1	1	0.82
C_1	string.each_char { |c|	1	1	1	1	1	1	0.82
C_2	if c===/[A-Z]/	1	1	1	1	0	1	0.89
C_3	**let += 2**	1	1	1	1	0	0	1.00
C_4	elsif c===/[a-z]/	1	1	1	1	0	1	0.89
C_5	let += 1	1	1	0	0	0	0	0.71
C_6	elsif c===/[0-9]/	1	1	1	1	0	1	0.89
C_7	dig += 1	0	1	0	1	0	0	0.71
C_8	elsif not c===/[a-zA-Z0-9]/	1	0	1	0	0	1	0.58
C_9	other += 1 }	1	0	1	0	0	1	0.58
C_{10}	return let, dig, other	1	1	1	1	1	1	0.82
	end							
	Output vector (verdicts)	1	1	1	1	0	0	

that are in v_2 but not in v_1. Table I illustrates the use of SFL for a function which counts the characters in a string, which is tested by test cases t_1 to t_6. The SC column shows the similarity coefficient, calculated against the output vector, for each line of code. In this example, line C_3 is the line most likely to be faulty as it has the highest SC. In this case, it is clear to see that this is correct as `let` should be increased by 1 instead of 2. In the remainder of this paper, we present our approach for using spectrum-based analysis for detecting performance regressions.

V. 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 to 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 behaviour of an application during the execution of a certain test execution in a *profile*. After an update, we compare the behaviour of our application during the execution of the same test using that profile.

A. 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 (see Section VI) all functions in our application that perform I/O 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, due to 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 tradeoff 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:

TABLE II
ILLUSTRATION OF THE PROFILE GENERATION IDEA

Revision: 1	Avg. # 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 III
ILLUSTRATION OF PROFILE COMPARISON

Revision: 2	Average # 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()	10000	12000	8000	0	0	0	0
Output vector				1	1	1	

- For every function:
  - Calculate the average number of bytes 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, this will result in a number. Define the highest and lowest values for this number as the accepted range for that revision

Table II demonstrates this idea. The profile can be read as: '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.'

B. 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 done manually, this is a tedious process and prone to mistakes. As explained in Section IV, SFL is a technique which closely resembles the human diagnosis process. Therefore, we propose to automate the comparison using a spectrum-based technique. 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 apply spectrum-based analysis to performance.

For every test execution t_i, we record the I/O write data as described in Section V-A. After this, 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 get a binary vector in which every row represents a function. If we place those vectors next to each other, we get a matrix looking similar to the activity matrix described in Section V-A. Table III shows sample data and the resulting matrix for three test executions t_i, after comparing them with the profile of Table II. We use three executions here for brevity, but this may be any number.

The analysis step now works as follows. When performance did not change after the source code update, 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 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). Using the SC, we can make a ranking of the functions most likely to have been affected by the update. When all SC's are close or equal to 1, the average number of bytes written did not change for any function after the update. [2] The functions with SC closer to 0 are likely to have been affected by the update. In Table III, from the SC column we can conclude that the performance of the `generateReport()` and `writeCache()` functions were 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 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 bytes) or larger (e.g., 50 kilobytes). 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 test executions. A final extension we make to our report is that we collect data for a complete stack trace instead of a single function. The main reasons for this are that 1) the behaviour of a function may be defined by the origin from which it was called (e.g., a database `commit()` function) and 2) this makes the optimization process easier, as we have a more precise description of the function behaviour.

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 behaviour most likely has changed due to 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 which 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 IV 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 to 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 to 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 5MB, which may be high or low, depending on the test suite and the type of application.

VI. Implementation

In this section, we present the implementation of our approach called SPECTRAPERF. SPECTRAPERF is part of the open-source experiment runner framework GUMBY[3], and is available for download from the GUMBY repository. Our implementation consists of two parts, the data collection and the data processing part.

A. 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 which use libraries written in different languages. In addition, by instrumenting system calls, we can monitor data which is 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 system call to actually write the file. Hence, if we could instrument that system call and find out from which application-level function it was called, we can obtain the application-level information with much less effort.

[2]Note that this terminology is different from that in Section IV, in which a SC close to 1 means the component is likely to be faulty. We do not use the terminology 'faulty', as the effect of an update may be positive or negative. Hence, in this case we feel the more intuitive explanation of a high SC is the high similarity compared to the previous version.

[3]http://www.github.com/tribler/gumby

TABLE IV
SIMILARITY REPORT FOR TABLE III

Revision: 2						
Function	SC	# calls	`Impact`	`TotalImpact`	`RangeDifference`	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

By combining application-level and operating system-level data with Systemtap, we can get a detailed profile of the writing behaviour of our application and any libraries it uses. Systemtap allows dynamic instrumentation [11] by writing *probes* which 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 which writes to a file. Note that, if we want to monitor other metrics such as memory usage, we must probe other system calls[4].

The subject system of our user study (see Section VII), 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 [5]. By running these probes together with any Python application, we can monitor write I/O usage on a function-level granularity.

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

Listing 1. Set of available probe points in a writing Python function.

```
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 */
}
```

Listing 2. Description of probes for monitoring Python I/O write usage.

While Systemtap natively supports C and C++, 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 we enter (*python.function.entry*), we add the function name to an array for that thread. Then, for all the writes done during the execution of that function, we sum the total number of bytes written per file (*syscall.open.entry* and *syscall.write.entry*). 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.

B. Data Processing

After collecting the data, we import it into a SQLite[6] database using R[7] and Python. From this database, we generate a report for each test execution (the *test execution report*) which shows:

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.

VII. DESIGN OF THE FIELD USER STUDY

We evaluate our approach in a field user study. The goal of our study is to determine whether performance bottlenecks 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 which 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 100 KLOC. Tribler uses Dispersy [12] as a fully decentralized solution for synchronizing messages over the network. Tribler has been under

[4]See http://asm.sourceforge.net/syscall.html for a (partial) list of system calls in Linux
[5]See the GUMBY source code for the exact implementation.
[6]http://www.sqlite.org/
[7]http://www.r-project.org/

development for 9 years. As a result, all 'low-hanging fruit' performance optimizations have been found with the help of traditional performance analysis tools. 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 minute idle run of Tribler, in which Tribler is started without performing any actions in the GUI. However, because of the peer-to-peer 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 programmer 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 [8]. We implemented a script in GUMBY which does the following for each of the last n commits:

1) Execute the required test 5 times[9], together with the Systemtap probes
2) Load the monitored data into a SQLite database
3) Generate a test execution report for each test execution as explained in Section VI-B
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 IV
7) Generate a profile to compare with the next revision

After all commits have been analyzed, the data is 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 step 3 and 6, i.e., 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[9]. In the Tribler case study, we will analyze a 10 minute idle run of Tribler for the last 100 revisions[9]. Tribler needs some time to shutdown. If for some reason, Tribler does not shutdown by itself, the instance is killed after 15 minutes using a process guard.

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:

[8]http://www.github.com/tribler

[9]Note that these numbers were chosen based on the execution time of the tests. We have no statistical evidence that this is indeed an optimal value.

TABLE V
OVERVIEW OF DISPERSY EVALUATION RESULTS

Phenomenon	Participant	# 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	Yes
	II	1	Yes

1) To select the 3 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 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 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.

VIII. EVALUATION

A. Case Study I: Dispersy Unit Test Suite

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, due to 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 which resulted in more I/O, hence he selected different phenomena to investigate. Next, we discuss each phenomenon and the way the participants investigated them. Table V gives an overview of which ranked position the participants analyzed and whether the information provided was useful.

1) Phenomenon A: The increase was caused by a bugfix. Before this bugfix, data was 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.

2) Phenomenon B: The drop in writes is due to 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

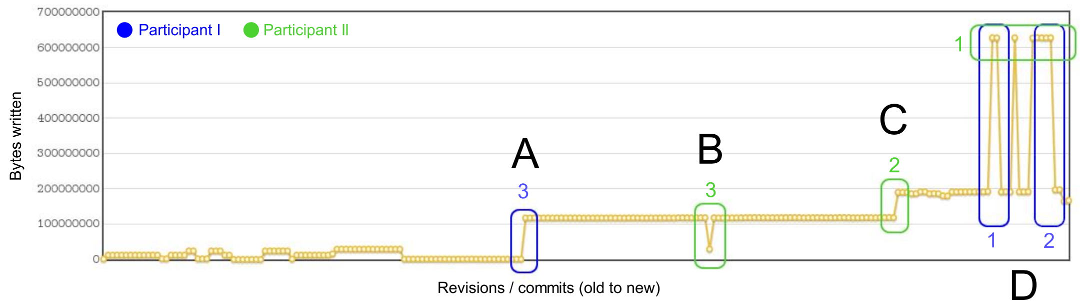

Fig. 1. Average number of bytes written during an execution of the Dispersy unit test suite for each commit

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.

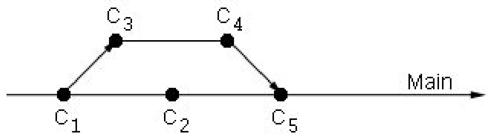

Fig. 2. Order of traversal of commits in git log (C_1 to C_5)

Likewise, the drop during phenomenon B was caused by testing 'old' code, which lead to a confusing report. This can be avoided 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.

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.

3) Phenomenon C: In the updated code, a different test class was used which 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 #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 #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 #1 ranked stack trace by itself, we marked this stack trace as 'partly useful' in Table V. 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 to the previous revision.

4) 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 message is committed to the database, had a SC of 0 and a `TotalImpact` of 435MB. As the number of calls was 10k, the issue was easy to fix for the participants. The fix was verified using our approach. From the graph, we could see that the total writes decreased from 635MB to approximately 200MB. From the similarity report, we could see that the number of calls to the stack trace decreased from 10k to 8.

TABLE VI
OVERVIEW OF TRIBLER EVALUATION RESULTS

Phenomenon	Participant	# Ranking	Helpful?
A	I	45	Yes
B	II	-	No
C	I	1	No
	I	65	Yes
	II	-	No
D	I	1	Yes
	II	24, 26, 27	No
	II	17, 31, 2	Yes

B. Case Study II: Tribler Idle Run

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 behaviour during the idle run (i.e., just starting the application) is influenced by external factors in Tribler. Due to its decentralized nature, an idle client may still be facilitating searches or 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 VI summarizes these results for the Tribler case study.

1) Phenomenon A: During 2 out of 5 test executions, Tribler crashed for this commit. Hence, less 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.

Participant's Analysis: From the reports, participant I was able to detect that less messages were received, but he was not able to detect the actual cause for this. Therefore, he granted the behaviour to noise due to external factors.

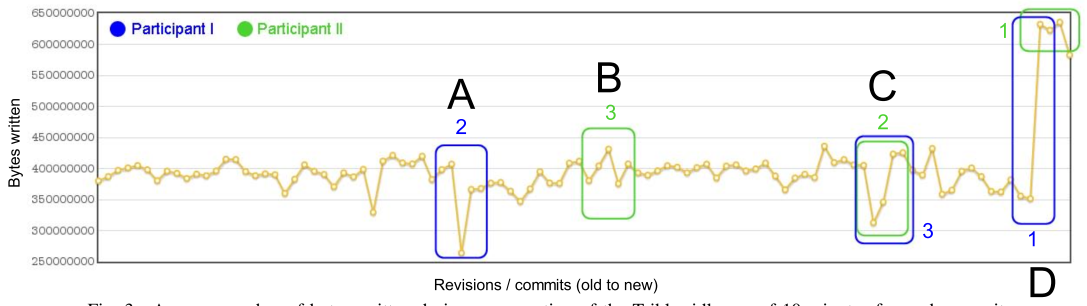

Fig. 3. Average number of bytes written during an execution of the Tribler idle run of 10 minutes for each commit

2) Phenomenon B: No significant changes were found, the variation was due to external factors.

Participant's Analysis: Participant II correctly diagnosed this as noise due to external factors.

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

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

4) Phenomenon D: The reason for the large increase in writes is that the committed code made part of Tribler crash. As a result, the idle run had to be killed after 15 minutes 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 which caused the large number of writes. They did not directly indicate the partial crash as the cause. Both participants advised to include 1) the actual duration of the execution and 2) 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 resources. 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 340MB 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.

C. Evaluation Results

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

Even so, the participants were able to correctly analyze and diagnose 3 out of 4 phenomena in the Dispersy report and 2 out of 4 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 VII. 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
3) The total duration of the test execution
4) An explanation of (or solution to) the 'git log order' effect, explained in Section VIII-A

After the user study, two phenomena out of two that could be optimized, were optimized after the case study based on our reports. In addition, both of these optimizations 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 bugfix. Finally, according to the participants, four out of the five phenomena which did not represent a performance regression, were easy to diagnose.

In Table V and VI, 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 3 rank as well. Hence, we can conclude that the ranking given by our approach is useful after a small adjustment. An observation we made was that the participants all used the `TotalImpact` as a guideline for indicating whether the change in behaviour of a stack trace was significant enough to investigate further. After this, they checked the SC to see in how many test executions the behaviour 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.

IX. Discussion

A. The Evaluation Research Questions Revisited

1) Does our approach provide enough information to detect performance regressions?: From our evaluation, we conclude that our reports provide, after adding the information explained in Section VIII-C, enough information for detecting performance regressions. In our study, two out of two detected

TABLE VII
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	Yes
C	I	-	C	I	Yes
	II	Yes		II	Yes
D	I	Yes	D	I	Partly
	II	Yes		II	Partly

regressions were diagnosed correctly by the participants.

2) Does our approach provide enough information to guide the performance optimization process?: Our evaluation showed that our approach provides enough information for guiding the performance optimization process as this user study alone resulted in two optimizations (Dispersy phenomenon D and Tribler phenomenon D) that have immediately been carried through in the respective projects.

3) Does our approach provide enough information to verify the effect of made performance optimizations?: Our approach provides enough information to validate the two optimizations 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 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, we will add this in future work.

4) How does our approach work for test executions which are influenced by external factors?: From our Tribler case study, we get an indication that our approach can deal with influence from external factors, as the participants were able to completely explain 2 out of 4 performance phenomena and partly explain the remaining 2. However, the results should be treated with more care than for a test which is not influenced by external factors, as they are more likely to represent noise due to those factors. In future work, we will do research on how we can minimize the effect of external factors.

B. Scalability & Limitations

For the moment, the overhead of our approach is considerable, mostly due to the monitoring by Systemtap. However, to the best of our knowledge, Systemtap is the only available option for monitoring Python code with such granularity. In addition, our approach is meant to run in testing environments and as we do not take execution time into account in our analysis, we do not see overhead as a limitation.

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.

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.

C. Threats to Validity

We have performed our field study on an application which has been under development for 9 years and is downloaded over a million times. This application is well-developed and 'low-hanging fruit' optimizations are already done, because of the importance of performance for Tribler due to its peer-to-peer nature. The user study was carried out with developers who have considerable experience with the application.

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, due to the low number of test executions, we feel that using a value such as the standard deviation does not add to the reliability of the profiles. In addition, our evaluation shows that we can achieve good results with this approach.

A threat to the validity of our evaluation is that we tested all commits instead of just the merge commits. As a result, we encountered crashing code more often, as these commits do not necessarily provide working code. In addition, it added some phenomena which are difficult to explain without knowing this effect (see Section VIII). However, after making the participants aware of this effect, they both agreed it would be easy to detect in future investigations.

X. Related Work

Spectrum-based analysis has been successfully used before for fault localization [6], [9]. To the best of our knowledge, we are the first to apply spectrum-based analysis to performance.

Comparison of execution profiles and the detection of performance regressions have received surprisingly little attention in research. Savari [13] has proposed a method which works for frequency-based profiling methods. Our approach works for any type of metric on a function-level granularity.

Bergel et al. [14] have proposed a profiler for Pharo which compares profiles using visualization. In their visualization, the size of an element describes the execution time and number of calls. Alcocer [15] extends Bergel's approach by proposing a method for reducing the generated callgraph. These visualizations require human interpretation, which is difficult when the compared profiles are very different [14]. Our approach provides a textual ranking, which we expect to be easier to interpret. However, we believe that the work of Bergel et al., Alcocer and our approach can be supplemental to each other, and we will investigate this in future work.

Foo et al. [16] present an approach for detecting performance regressions by mining performance repositories. Nguyen et al. [17] propose an approach for detecting performance regressions using statistical process control techniques. In contrast to our approach, these approaches do not give information on a function-level granularity, but focus on reporting on differences in system-level metrics instead.

Horky et al. [18] and Heger et al. [19] propose approaches for integrating performance tests into the unit test suite. Their approaches require the generation of new unit tests, while our approach can be attached to existing test suites.

XI. Conclusion

In this paper, we proposed a technique for detecting and analyzing performance regressions using a spectrum-based approach. 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 optimizing write I/O, but our approach can easily be extended to other metrics such as read I/O, memory and CPU usage by changing the monitoring component.

In a field user study, we showed that our approach provides adequate information to detect performance regressions and guides the performance optimization process. In fact, our field user study resulted in two optimizations made to our subject system. To summarize, we make the following contributions:

1) An approach for the detection and analysis of performance regressions
2) An open-source implementation of this approach, called SpectraPerf
3) A field user study in which we show that our approach guides the performance optimization process

Revisiting our research questions:

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

How can we compare the generated performance profiles? We have presented our approach for using a spectrum-based technique to compare performance profiles, and provide a ranking of the stack traces which were most likely to have changed behaviour. This ranking is made based on the *similarity coefficient* compared to the previous performance profile, and the *total impact* of a source code change on performance. In our user study, we showed the ranking was useful in 6 out of 8 cases and helped the participants find two optimizations.

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 peer-to-peer client Tribler and one of its components, Dispersy. The field user study resulted in two optimizations, 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 3 out of 4 performance phenomena encountered during such a test using our approach. We also received indication that it works well for a test which was influenced by external factors, as the participants were able to explain 2 out of 4 performance phenomena completely and could partly explain the remaining 2 for such a test.

How can we guide the performance optimization process by doing performance regression tests? We have showed that our approach for spectrum-based performance analysis can guide the performance optimization process by detecting performance regressions. The results of our field user study alone, resulted in two optimizations to Tribler and Dispersy.

In future work, we will focus 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] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529–551, 1996.

[2] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[3] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[4] S. P. Reiss, "Visualizing the java heap to detect memory problems," in *Int'l Workshop Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 2009, pp. 73–80.

[5] M. Woodside, G. Franks, and D. Petriu, "The future of software performance engineering," in *Future of Softw. Engineering (FOSE)*. IEEE, pp. 171–187.

[6] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[7] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. Epema, M. Reinders, M. R. Van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 2, pp. 127–138, 2008.

[8] J. Larres, A. Potanin, and Y. Hirose, "A study of performance variations in the mozilla firefox web browser," in *Proc. Australasian Comp. Science Conference (ACSC)*. Australian Computer Society, Inc., 2013, pp. 3–12.

[9] C. Chen, H.-G. Gross, and A. Zaidman, "Spectrum-based fault diagnosis for service-oriented software systems," in *Int'l Conf. Service-Oriented Computing and Applications (SOCA)*. IEEE, 2012, pp. 1–8.

[10] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2006, pp. 39–46.

[11] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Proc. Ottawa Linux Symposium*, 2005, pp. 49–64.

[12] N. Zeilemaker, B. Schoon, and J. Pouwelse, "Dispersy bundle synchronization," TU Delft, Tech. Rep. PDS-2013-002, 2013.

[13] S. A. Savari and C. Young, "Comparing and combining profiles," *Journal of Instruction-Level Parallelism*, vol. 2, 2000.

[14] A. Bergel, F. Bañados, R. Robbes, and W. Binder, "Execution profiling blueprints," *Softw., Pract. Exper.*, vol. 42, no. 9, pp. 1165–1192, 2012.

[15] J. P. S. Alcocer, "Tracking down software changes responsible for performance loss," in *Proc. Int'l Workshop on Smalltalk Technologies (IWST)*. ACM, 2012, pp. 3:1–3:7.

[16] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Int'l Conf. Quality Software (QSIC)*. IEEE, 2010, pp. 32–41.

[17] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proc. ACM/SPEC Int'l Conf. on Performance Engineering (ICPE)*, 2012, pp. 299–310.

[18] V. Horky, F. Haas, J. Kotrc, M. Lacina, and P. Tuma, "Performance regression unit testing: A case study," in *Computer Performance Engineering*, ser. LNCS. Springer, 2013, vol. 8168, pp. 149–163.

[19] C. Heger, J. Happe, and R. Farahbod, "Automated root cause isolation of performance regressions during software development," in *Proc. ACM/SPEC Int'l Conf. Performance Engineering (ICPE '13)*, pp. 27–38.

TUD-SERG-2013-020
ISSN 1872-5392