An Automated Approach to Estimating Code Coverage Measures via Execution Logs

Boyuan Chen
York University
Toronto, Canada
chenfsd@cse.yorku.ca

Jian Song
Baidu Inc.
Beijing, China
songjian02@baidu.com

Peng Xu
Baidu Inc.
Beijing, China
xupeng@baidu.com

Xing Hu
Baidu Inc.
Beijing, China
huxing@baidu.com

Zhen Ming (Jack) Jiang
York University
Toronto, Canada
zmjiang@cse.yorku.ca

ABSTRACT

Software testing is a widely used technique to ensure the quality of software systems. Code coverage measures are commonly used to evaluate and improve the existing test suites. Based on our industrial and open source studies, existing state-of-the-art code coverage tools are only used during unit and integration testing due to issues like engineering challenges, performance overhead, and incomplete results. To resolve these issues, in this paper we have proposed an automated approach, called *LogCoCo*, to estimating code coverage measures using the readily available execution logs. Using program analysis techniques, LogCoCo matches the execution logs with their corresponding code paths and estimates three different code coverage criteria: method coverage, statement coverage, and branch coverage. Case studies on one open source system (HBase) and five commercial systems from Baidu and systems show that: (1) the results of LogCoCo are highly accurate (> 96% in seven out of nine experiments) under a variety of testing activities (unit testing, integration testing, and benchmarking); and (2) the results of LogCoCo can be used to evaluate and improve the existing test suites. Our collaborators at Baidu are currently considering adopting LogCoCo and use it on a daily basis.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

software testing, logging code, test coverage, empirical studies, software maintenance;

ACM Reference Format:
Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. 2018. An Automated Approach to Estimating Code Coverage Measures via Execution Logs. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238214

1 INTRODUCTION

A recent report by Tricentis shows that software failure caused $1.7 trillion in financial losses in 2017 [48]. Therefore, software testing, which verifies a system's behavior under a set of inputs, is a required process to ensure the system quality. Unfortunately, as software testing can only show the presence of bugs but not their absence [15], complete testing (a.k.a., revealing all the faults) is often not feasible [3, 36]. Thus, it is important to develop high quality test suites, which systematically examine a system's behavior.

Code coverage measures the amount of executed source code, when the system is running under various scenarios [18]. There are various code coverage criteria (e.g., statement coverage, condition coverage, and decision coverage) proposed to measure how well the test suite exercises the system's source code. For example, the statement coverage measures the amount of executed statements, whereas the condition coverage measures the amount of true/false decisions taken from each conditional statement. Although there are mixed results between the relationship of code coverage and test suite effectiveness [30, 42], code coverage is still widely used in research [53, 57, 59] and industry [1, 54, 58] to evaluate and improve the quality of existing test suites.

There are quite a few commercial (e.g., [14, 49]) and open source (e.g., [11, 32]) tools already available to automatically measure the code coverage. All these tools rely on instrumentation at various code locations (e.g., method entry/exit points and conditional branches) either at source code [6] or at binary/bytecode levels [11, 31]. There are three main issues associated with these tools when used in practice: (1) **Engineering challenges**: for real-world large-scale distributed systems, it is not straight-forward to configure and deploy such tools, and collect the resulting data [47, 71]. (2) **Performance overhead**: the heavy instrumentation process can introduce performance overhead and slow down the system execution [21, 29, 64]. (3) **Incomplete results**: due to various issues associated with code instrumentation, the coverage results from these tools do not agree with each other and are sometimes incomplete [27]. Hence, the application context of these tools are generally very limited (e.g., during unit and integration testing). It is very challenging to measure the coverage for the system under test (SUT) in a field-like environment to answer questions like evaluating the representativeness of in-house test suites [66]. Such problem is going to be increasingly important, as more and more systems are adopting the rapid deployment process like DevOps [40].

Execution logs are generated by the output statements (e.g., `Log.info(‘‘User’’ + user + ‘‘checked out’’)`) that developers insert into the source code. Studies have shown that execution logs have been actively maintained for many open source [8, 74] and commercial software systems [19, 55] and have been used extensively in practice for a variety of tasks (e.g., system monitoring [60], problem debugging [52, 73], and business decision making [4]). In this paper, we have proposed an approach, called *LogCoCo* (Log-based Code Coverage), which automatically estimates the code coverage criteria by analyzing the readily available execution logs. We first leverage program analysis techniques to extract a set of possible code paths from the SUT. Then we traverse through these code paths to derive the list of corresponding log sequences, represented using regular expressions. We match the readily available execution logs, either from testing or in the field, with these regular expressions. Based on the matched results, we label the code regions as `Must` (definitely covered), `May` (maybe covered, maybe not), and `Must-not` (definitely not covered) and use these labels to infer three types of code coverage criteria: method coverage, statement coverage, and branch coverage. The contributions of this paper are:

(1) This work systematically assesses the use of the code coverage tools in a practical setting. It is the first work, to the authors' knowledge, to automatically estimate code coverage measures from execution logs.
(2) Case studies on one open source and five commercial systems (from Baidu) show that the code coverage measures inferred by LogCoCo is highly accurate: achieving higher than 96% accuracy in seven out of nine experiments. Using LogCoCo, we can evaluate and improve the quality of various test suites (unit testing, integration testing, and benchmarking) by comparing and studying their code coverage measures.
(3) This project is done in collaboration with Baidu, a large scale software company whose services are used by hundreds of millions of users. Our industrial collaborators are currently considering adopting and using LogCoCo on a daily basis. This clearly demonstrates the usefulness and the practical impact of our approach.

Paper Organization: the rest of this paper is structured as follows. Section 2 explains issues when applying code coverage tools in practice. Section 3 explains LogCoCo by using a running example. Section 4 describes our experiment setup. Section 5 and 6 study two research questions, respectively. Section 7 introduces the related work. Section 8 discusses the threats to validity. Section 9 concludes the paper.

2 APPLYING CODE COVERAGE TOOLS IN PRACTICE

We interviewed a few QA engineers at Baidu regarding their experience on the use of the code coverage tools. They regularly used code coverage tools like JaCoCo [32] and Cobertura [11]. However, they apply these tools only during the unit and integration testing. It turned out that there are some general issues associated with these state-of-the-art code coverage tools, which limit their application contexts (e.g., during performance testing and in the field). We summarized them into the following three main issues, which are also problematic for other companies [47, 71]: (1) *Engineering challenges:* depending on the instrumentation techniques, configuring and deploying these tools along with the SUT can be tedious (e.g., involving recompilation of source code) and error-prone (e.g., changing runtime options). (2) *Performance overhead*: although these tools can provide various code coverage measures (e.g., statement, branch, and method coverage), they introduce additional performance overhead. Such overhead can be very apparent, when the SUT is processing hundreds or thousands of concurrent requests. Therefore, they are not suitable to be running during non-functional testing (e.g., performance or user acceptance testing) or in the field (e.g., to evaluate and improve the representativeness of the in-house test suites). (3) *Incomplete results*: the code coverage results from these tools are sometimes incomplete.

In this section, we will illustrate the three issues mentioned above through our experience in applying the state-of-the-art code coverage tools on HBase [23] in a field-like environment.

2.1 The HBase Experiment

HBase, which is an open source distributed NoSQL database, has been used by many companies (e.g., Facebook [50], Twitter, and Yahoo! [24]) serving millions of users everyday. It is important to assess its behavior under load (a.k.a., collecting code coverage measures) and ensure the representativeness of the in-house test suites (a.k.a., covering the behavior in the field).

YCSB [72] is a popular benchmark suite, originally developed by Yahoo!, to evaluate the performance of various cloud-based systems (e.g., Cassandra, Hadoop, HBase, and MongoDB). YCSB contains six core benchmark workloads (*A*, *B*, *C*, *D*, *E*, and *F*) which are derived by examining a wide range of workload characteristics from real-world applications [12]. Hence, we use this benchmark suite to simulate the field behavior of HBase.

Our HBase experiment was conducted on a three-machine-cluster with one master node and two region server nodes. These three machines have the same hardware specifications: Intel i7-4790 CPU, 16 GB memory, and 2 TB hard-drive. We picked HBase version 1.2.6 for this experiment, since it was the most current stable release by the time of our study. We configured the number of operations to be 10 million for each benchmark workload. Each benchmark test exercised all the benchmark workloads under one of the following three YCSB thread number configurations: 5, 10, and 15. Different number of YCSB threads indicates different load levels: the higher the number of threads, the higher the benchmark load.

2.2 Engineering Challenges

Since HBase is implemented in Java, we experimented with two Java-based state-of-the-art code coverage tools: JaCoCo [32] and Clover [10]. Both tools have been used widely in research (e.g., [27, 28, 41]) and practice (e.g., [47, 71]). These two tools use different instrumentation approaches to collecting the code coverage measures. Clover [10] instruments the SUT and injects its monitoring

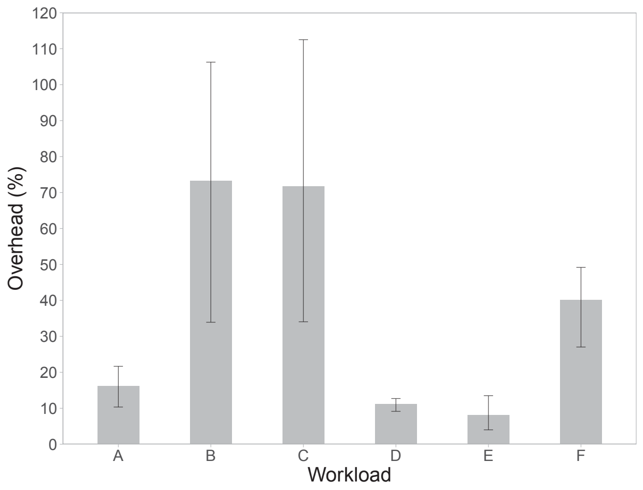

Figure 1: The JaCoCo overhead for the HBase experiment.

probes at the source code level, while JaCoCo [31] uses the bytecode instrumentation technique and injects its probes during runtime.

Overall, we found the configuration and the deployment processes for both tools to be quite tedious and error-prone. For example, to enable the code coverage measurement by JaCoCo, we had to examine various HBase scripts to figure out the command line options to startup HBase and its required jar files. This process was non-trivial and required manual effort, as the command line options could differ from systems to systems and even different versions of the same systems. The process for Clover was even more complicated, as we had to reconfigure the Maven build system to produce a new set of instrumented jar files. In addition, we could not simply copy and replace the newly instrumented jar files into the test environment due to dependency changes. It required a thorough cleanup of the test environment before re-deploying SUT and running any tests. We considered such efforts to be non-trivial, as we had to repeat this process on all three target machines. This effort could be much higher if the experiments were done on tens or hundreds of machines, which is considered as a normal deployment size for HBase [65].

We decided to proceed with JaCoCo. Its instrumentation and deployment process were less intrusive, as the behavior of HBase needed to be assessed in a field like environment.

2.3 Performance Overhead

We ran each benchmark test twice: once with JaCoCo enabled, and once without. We gathered the response time statistics for each benchmark run and estimated the performance overhead introduced by JaCoCo. Figure 1 shows the performance overhead for the six different workloads (workload $A, \ldots, F$). Within each workload, the figure shows the average performance overhead (in percentages) as well as the confidence intervals across different YCSB thread numbers. For example, the average performance overhead for workload A is 16%, but can vary from 10% to 22% depending on the number of threads.

Depending on the workload, the performance impact of JaCoCo varies. Workload B has the highest impact (79% to 106%) with JaCoCo enabled, whereas workload E has the smallest impact (4% to 13%). Overall, JaCoCo does have a negative impact on the SUT with a noticeable performance overhead (> 8% on average) across all benchmark tests. Hence, it is not feasible to deploy JaCoCo in a field-like environment with the SUT, as it can significantly degrade the user experience.

2.4 Incomplete Results

We sampled some of the code coverage data produced by JaCoCo for manual verification. We found that JaCoCo did not report the code coverage measures for some modules. JaCoCo only instrumented the HBase modules (a.k.a., the `hbase-server` module) in which the YCSB benchmark suite directly invoked. If the `hbase-server` module invokes another module (e.g., `client`) not specified during the HBase startup, the `client` module will not be instrumented by JaCoCo and will not have any coverage data reported. For example, during our experiment, the logging statement from the method `setTableState` in `ZKTableStateManager.java` was outputted. Hence, `setTableState` should be covered. Since `setTableState` calls `setTableStateInZK`, which calls `joinZNode` in `ZKUtil.java`, the method `joinZNode` should also be covered. However, the `joinZNode` method was marked as not covered by JaCoCo. A similar problem was also reported in [27].

To resolve the three issues mentioned above, we have proposed a new approach to automatically estimating the code coverage measures by leveraging the readily available execution logs. Our approach estimates the code coverage by correlating information in the source code and the log files, once the tests are completed. It imposes little performance overhead to the SUT, and requires no additional setup or configuration actions from the QA engineers.

3 LOGCOCO

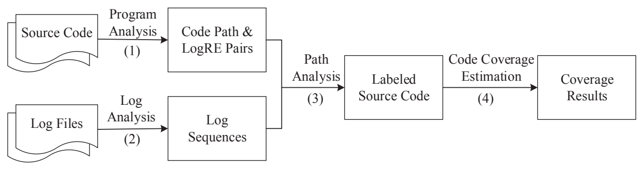

Figure 2: An overview of LogCoCo.

In this section, we will describe LogCoCo, which is an automated approach to estimating code coverage measures using execution logs. As illustrated in Figure 2, our approach consists of the following four phases: (1) during the program analysis phase, we analyze the SUT's source code and derive a list of possible code paths and their corresponding log sequences expressed in regular expressions (LogRE). (2) During the log analysis phase, we analyze the execution log files and recover the log sequences based on their execution context. (3) During the path analysis phase, we match each log sequence with one of the derived LogRE and highlight the corresponding code paths with three kinds of labels: `May`, `Must`, and `Must-not`. (4) Based on the labels, we estimate the values for the following three code coverage criteria: method coverage, statement coverage, and branch coverage. In the rest of this section, we will explain the aforementioned four phases in details with a running example shown in Figure 3.

Log Sequences		**Seq 1**	**Seq 2**	**Final Code Coverage**
Branch selection set		[if@4:*true*,for@12:*true*,if@16: *false*]	[if@4:*false*,for@12:*true*,if@16: *false*]	
LogRE		(Log@3)(Log@51)(Log@14)+	(Log@3)(Log@14)+	
Code Snippets		**Intermediate Code Coverage**		
1	`void computation(int a, int b) {`	Must	Must	Must
2	`  int a = randomInt();`	Must	Must	Must
3	`  log.info("Random No: " + a);`	Must	Must	Must
4	`  if (a < 10) {`	Must	Must	Must
5	`    a = process(a);`	Must	Must-not	Must
6	`  } else {`	Must-not	Must	Must
7	`    a = a + 10;`	Must-not	Must	Must
8	`  }`	-	-	-
9	`  if (a % 2 = 0) {`	Must	Must	Must
10	`    a ++;`	May	May	May
11	`  }`	-	-	-
12	`  for (;b < 3; b++) {`	Must	Must	Must
13	`    a ++;`	Must	Must	Must
14	`    log.info("Loop: " + a);`	Must	Must	Must
15	`  }`	-	-	-
16	`  if (a > 20) {`	Must	Must	Must
17	`    log.info("Check: " + a);`	Must-not	Must-not	Must-not
18	`  }`	-	-	-
19	`}`	-	-	-
⋮	⋮	⋮	⋮	⋮
50	`int process(int num) {`	Must	Must	Must
51	`  log.info("Process: " + (++num));`	Must	Must	Must
52	`  return num;`	Must	Must	Must
53	`}`	-	-	-

Figure 3: The code snippet of our running example.

3.1 Phase 1 - Program Analysis

Different sequences of log lines will be generated if the SUT executes different scenarios. Hence, the goal of this phase is to derive the matching pairs between the list of possible code paths and their corresponding log sequences. This phase is further divided into three steps:

Step 1 - Deriving AST for Each Method. we derive the per method Abstract Syntax Tree (AST) using a static analysis tool called Java Development Tools (JDT) [17] from the Eclipse Foundation. JDT is a very robust and accurate program analysis tool, which has been used in many software engineering research papers (e.g., bug prediction [78], logging code analysis [9], and software evolution [68]).

Consider our running example shown on the left part of Figure 3. This step will generate two ASTs for the two methods: `computation` and `process`. Each node in the resulting AST is marked with the corresponding line number and the statement type. For example, at line 3, there is a logging statement and at line 4 there is an if statement. There are also edges connecting two nodes if one node is the parent of the other node.

Step 2 - Deriving Call Graphs. the resulting ASTs from the previous step only contain the control flow information at the method level. In order to derive a list of possible code paths, we need to form call graphs by chaining the ASTs of different methods. We have developed a script, which automatically detects method invocations in the ASTs, and links them with the corresponding method body. In the running example, our script will connect the method invocation of the `process` method at line 5 with the corresponding method body starting at line 50.

Step 3 - Deriving Code Paths and LogRE Pairs. based on the resulting call graphs, we will derive a list of possible code paths. The number of resulting code paths depends on the number and the type of control flow nodes (e.g., if, else, for, and while), which may contain multiple branching choices. Consider the if statement at line 4 in our running example: depending on the true/false values for the conditional variable `a`, there can be two call paths generated: `[4, 5, 50, 51, 52]` and `[4, 6, 7]`.

We leverage the Breadth-First-Search (BFS) algorithm to traverse through the call graphs in order to derive the list of possible code paths and their corresponding LogREs. When visiting each control flow node, we pick one of the decision outcomes for that node and go to the corresponding branches. During this process, we also keep track of the resulting LogREs. Each time when a logging statement is visited, we add it to our resulting LogRE. If a logging statement is inside a loop, a "+" sign will be appended to it indicating that this logging statement could be printed more than once. For the logging statement, which is inside a conditional branch within a loop, it will be appended with a "?" followed by a "+". In the end, we will

```
1  2018-01-18 15:42:18,158 INFO   [Thread-1] [Test.java:3] Random No: 2
2  2018-01-18 15:42:18,159 INFO   [Thread-2] [Test.java:3] Random No: 4
3  2018-01-18 15:42:18,159 INFO   [Thread-1] [Test.java:51] Process: 3
4  2018-01-18 15:42:18,162 INFO   [Thread-2] [Test.java:14] Loop: 6
5  2018-01-18 15:42:18,163 INFO   [Thread-1] [Test.java:14] Loop: 4
6  2018-01-18 15:42:18,163 INFO   [Thread-1] [Test.java:14] Loop: 5
```

Figure 4: Log file snippets for our running example.

generate a branch selection set for this particular code path and its corresponding LogRE. There can be some control flow nodes, under which there is no logging statement node. In this case, we cannot be certain if any code under these control flow nodes will be executed. For scalability concerns, we do not visit the subtrees under these control flow nodes.

In our running example, there are four control flow nodes: line 4 (`if`), 9 (`if`), 12 (`for`), and 16 (`if`). If the conditions are true for line 4 and 12, and false for line 16, the branch selection set is represented as `[if@4:true, for@12:true, if@16: false]`. The value for the `if` condition node at line 9 is irrelevant, as there is no logging statement under it. We will not visit its subtree, as no changes will be made to the resulting LogRE. The resulting code path for this branch selection is `1,2,3,4,5,50,51,52,9,(10),12,13,14,16`. The corresponding LogRE is `(Log@3)(Log@51)(Log@14)+`. Line 10 in the resulting code path is shown in brackets, because there is no logging statement under the `if` condition node at line 9. Thus, we cannot tell if line 10 is executed based on the generated log lines.

3.2 Phase 2 - Log Analysis

Execution logs are generated when logging statements are executed. However, since there can be multiple scenarios executed concurrently, logs related to the different scenario executions may be inter-mixed. Hence, in this phase, we will recover the related logs into sequences by analyzing the log files. Suppose after executing some test cases for our running example, a set of log lines, shown in Figure 4, is generated. This phase is further divided into the following three steps:

Step 1 - Abstracting Log Lines. each log line contains static texts, which describe the particular logging context, and dynamic contents, which reveal the SUT's runtime states. Logs generated by modern logging frameworks like Log4j [45] can be configured to contain information such as file name and line number. Hence, we can easily map the generated log lines to the corresponding logging statements. In our example, each log line contains the file name `Test.java` and the line number. In other cases, if the file name and the line number is not printed, we can leverage existing log abstraction techniques (e.g., [25, 34]), which automatically recognize the dynamically generated contents and map log lines into the corresponding logging statements.

Step 2 - Grouping Related Log Lines. each log line contains some execution contexts (e.g., thread or session or user IDs). In this step, we group the related log lines into sequences by leveraging these execution contexts. In our running example, we group the related log lines by their thread IDs. There are in total two log sequences in our running example, which correspond to `Thread-1` and `Thread-2`.

Step 3 - Forming Log Sequences. in this step, we replace the grouped log line sequences into sequences of logging statements. For example, the log line sequence of line `1,3,5,6` grouped under `Thread-1` becomes `Log@3, Log@51, Log@14, Log@14`.

3.3 Phase 3 - Path Analysis

Based on the obtained log line sequences from the previous phase, we intend to estimate the covered code paths in this phase using a four-step process.

Step 1 - Matching Log Sequences with LogREs. we match the sequences of logging statements obtained in Phase 2 with the LogREs obtained in Phase 1. The two recovered log sequences are matched with the two LogREs, which are shown on the third row in Figure 3. The sequence of logging statement in our running example `Log@3, Log@51, Log@14, Log@14` (a.k.a., *Seq 1*) will be matched with LogRE `(Log@3)(Log@51)(Log@14)+`.

Step 2 - Labeling Statements. in the second step, based on each of the matched LogRE, we apply three types of labels to the corresponding source code based on their estimated coverage: `Must`, `May`, and `Must-Not`. The lower part of Figure 3 shows the results for our working example. For *Seq 1*, we label lines `1,2,3,4,5,9,12,13,14,16, 50,51,52` as `Must`, as these lines are definitely covered if the above log sequence is generated. Line 10 is marked as `May`, because we are uncertain if the condition of the `if` statement at line 9 is satisfied. Lines `6,7,17` are marked as `Must-Not`, because the branch choice is true at line 4 and false at line 17. Due to the page limit, we do not explain the source code labeling process for *Seq 2*.

Step 3 - Reconciling Statement-level Labels. as one line of source code may be assigned with multiple different labels from different log sequences, in the third step, we reconcile the labels obtained from different log sequences and assign one final resulting label to each line of source code. We use the following criteria for our assignment:

- **At least one `Must` label**: since a particular line of source code is considered as "covered", when it has been executed at least once. Hence, if there is at least one `Must` label assigned to that line of source code, regardless of other scenarios, it is considered as covered (a.k.a., assigning `Must` labels as the final resulting label). In our running example, line 5 is marked as `Must` in *Seq 1* and `Must-not` in *Seq 2*. Therefore, it will be marked as `Must` in the final label.
- **No `Must` labels, and at least one `May` label**: in this particular case, we may have a specific line of source code assigned with all `May` labels, or a mixture of `May` and `Must-not` labels. As there is a possibility that this particular line of source code can be covered by some test cases, we assigned it to be `May` in the final resulting label. In our running example, line 10 is marked `May`.
- **All `Must-not` labels**: in this particular case, since there are no existing test cases covering this line of source code, we assigned it to be `Must-not` in the final resulting label. In our running example, line 17 is marked `Must-not` in both *Seq 1* and *Seq 2*. It will be marked as `Must-not` in the final label.

Step 4 - Inferring Labels at the Method Levels. based on the line-level labels, we assign one final label to each method using the following criteria:

- for a particular method, if there is at least one line of source code labeled as `Must`, this method will be assigned with a `Must` label. In our running examples, both methods will be labeled as `Must`.
- for methods without `Must` labeled statements, we apply the following process:
  - **Initial Labeling**: all of the logging statements under such methods should already be labeled as `Must-not`, since none of these logging statements are executed. If there is at least one logging statement which is not under any control flow statement nodes in the call graph, this method will be labeled as `Must-not`.
  - **Callee Labeling**: starting from the initial set of the `Must-not` labeled methods, we search for methods that will only be called by these methods, and assign them with the `Must-not` labels. We iteratively repeat this process until no more methods can be added to the set.
  - **Remaining Labeling**: we assign the `May` labels to the remaining set of unlabeled methods.

Similarly, each branch will be assigned with one final resulting label based on the statement-level labels. Due to space constraints, we will not explain the process here.

Phase 4 - Code Coverage Estimation

In this phase, we estimate the method/branch/statement-level code coverage measures using the labels obtained from the previous phase. As logging statements are not instrumented everywhere, there are code regions labeled as `May`, which indicates uncertainty of coverage. Hence, when estimating the code coverage measures, we provide two values: a minimum and a maximum value for the above three coverage criteria. The minimum value of statement coverage is calculated as $\frac{\#\ of\ \texttt{Must}\ labels}{Total\#\ of\ labels}$, and the maximum value is calculated as $\frac{\#\ of\ \texttt{Must}\ labels + \#\ of\ \texttt{May}\ labels}{Total\#\ of\ labels}$. In our running example, the numbers of `Must`, `May`, and `Must-not` statements are 15, 1, and 1, respectively. Therefore, the range of the statement coverage is from 88% ($\frac{15}{15+1+1} \times 100\%$) to 94% ($\frac{15+1}{15+1+1} \times 100\%$).

Similarly, since the number of `Must`, `May`, and `Must-not` branches is 5, 1, and 2 in our running example, the range of branch coverage is from 62.5% ($\frac{5}{5+1+2} \times 100\%$) to 87.5% ($\frac{5+2}{5+1+2} \times 100\%$).

The number of `Must`, `May`, and `Must-not` methods are 2, 0, and 0. Hence, the method level coverage is 100%.

4 CASE SETUP

To evaluate the effectiveness of our approach, we have selected five commercial projects from Baidu and one large-scale open-source project in our case study. Table 1 shows the general information about these projects in terms of their project name, project descriptions, and their sizes. All six projects are implemented in Java and their domains span widely from web services, to application platforms and NoSQL databases. The main reason why we picked commercial projects to study is that we can easily get hold of QA engineers for questions and feedback. The five commercial projects (C_1, C_2, C_3, C_4, and C_5) were carefully selected based on consultations with Baidu's QA engineers. We also picked one large-scale popular open source project, HBase [23], because we can freely discuss about the details. We focus on HBase version 1.2.6 in this study, since it is the most recent stable release by the time of the study. All six studied projects are actively maintained and being used by millions or hundreds of millions of users worldwide. We proposed the following two research questions (RQs), which will be discussed in the next two sections:

Table 1: Information about the six studied projects.

Projet	Descriptions	LOC
C_1	Internal API library	24K
C_2	Platform	80K
C_3	Cloud service	12K
C_4	Video streaming service	35K
C_5	Distributed file system	228K
HBase	Distributed NoSQL Database	453K

- **RQ1: (Accuracy)** *How accurate is LogCoCo compared to the state-of-the-art code coverage tools?* The goal of this RQ is to evaluate the quality of the code coverage measures derived from LogCoCo against the state-of-the-art code coverage tools. We intend to conduct this study using data from various testing activities.
- **RQ2: (Usefulness)** *Can we evaluate and improve the existing test suites by comparing the LogCoCo results derived from various execution contexts?* The goal of this RQ is to check if the existing test suites can be improved by comparing the estimated coverage measures using LogCoCo from various system execution contexts.

5 RQ1: ACCURACY

On one hand, existing state-of-the-art code coverage tools (e.g., Jacoco [32], Cobertura [11]) collect the code coverage measures by excessively instrumenting the SUT either at the source code [6] or at the binary/bytecode levels [11, 31]. The excessive instrumentation (e.g., for every method entry/exit, and for every conditional and loop branch) ensures accurate measurements of code coverage, but imposes problems like deployment challenges and performance overhead (Section 2), which limit their application context. On the other hand, LogCoCo is easy to setup and imposes little performance overhead by analyzing the readily available execution logs. However, the estimated code coverage measures may be inaccurate or incomplete, as developers only selectively instrument certain parts of the source code by adding logging statements. Hence, in this RQ, we want to assess the quality of the estimated code coverage measures produced by LogCoCo.

5.1 Experiment

We ran nine test suites for the six studied projects as shown in Table 2. The nine test suites contained unit and integration tests. Since the unit test suites were not configured to generate logs for C_1, C_4, and C_5, we did not include them in our study. C_5 and HBase are distributed systems, so we conducted their integration tests in a field-like deployment setting.

Each test suite was run twice: once with the JaCoCo configured, and once without. We used the code coverage data from JaCoCo as our oracle and compared it against the estimated results from LogCoCo. For all the experiments, we collected data like generated log files and code coverage measures from JaCoCo. JaCoCo is a widely used state-of-the-art code coverage tool, which is used in both research [27, 28, 41] and practice [47, 71]. We picked JaCoCo to ensure that the experiments could be done in a field-like environment. Code coverage tools, which leverage source code-level instrumentation techniques, require recompilation and redeployment of the SUT. Such requirements would make the SUT's testing behavior no longer closely resemble the field behavior. JaCoCo, a bytecode instrumentation based code coverage tool, is less invasive and instruments the SUT during runtime. For each test, we gathered the JaCoCo results and the log files. Depending on the tests, the sizes of the log files range from 8 MB to 1.1 GB.

5.2 Data Analysis

We compared the three types of code coverage measures (method, statement, and branch coverage) derived from LogCoCo and JaCoCo. Since LogCoCo marks the source code for each type of coverage using the following three labels: `Must`, `May`, and `Must-not`, we calculated the percentage of correctly labeled entities for three types of labels.

For the `Must` labeled methods, we calculated the portion of methods which are marked as *covered* in the JaCoCo results. For example, if LogCoCo marked five methods as `Must` among which four were reported as *covered* in JaCoCo, the accuracy of the LogCoCo method-level coverage measure would be $\frac{4}{5} \times 100\% = 80\%$. Similarly, for the `Must-not` labeled entities, we calculated the percentage of methods which were marked as *not covered* by JaCoCo.

For the `May` labeled methods, we calculated the portion of methods which are reported as *covered* by JaCoCo. Note that this calculation is ***not*** to assess the accuracy of the `May` covered methods, but to assess the actual amount of methods which are indeed covered during testing.

When calculating the accuracy of the statement and branch level coverage measures from LogCoCo, we only focused on code blocks from the `Must` covered methods. This is because all the statement and branch level coverage measures will be `May` or `Must-not` for the `May` or `Must-not` labeled methods, respectively. It would not be meaningful to evaluate these two cases again at the statement or branch level.

The evaluation results for the three coverage measures are shown in Table 2. If a cell is marked as "-", it means there is no source code assigned with the label. We will discuss the results in details below.

5.3 Discussion on Method-Level Coverage

As shown in Table 2, all methods labeled with `Must` are 100% accurate. It means that LogCoCo can achieve 100% accuracy when detecting covered methods. Rather than instrumenting all the methods like the existing code coverage tools do, LogCoCo uses program analysis techniques to infer the system execution contexts. For example, only 13% of the methods in C_1 have logs printed. The remaining 87% of the `Must` covered methods are inferred indirectly.

The methods labeled with `Must-not` are not always accurate. Three commercial projects (C_3, C_4, and C_5), and HBase have some methods which are actually covered in the tests but are falsely flagged as `Must-not` covered methods. Except for three cases, the accuracy of the `Must-not` labeled methods are all above 90%. We manually examined the misclassified instances and found the following two main reasons: (1) we have limited the size of our call graphs to 20 levels deep or a maximum of 100, 000 paths per AST tree due to memory constraints of our machine. Therefore, we missed some methods, which had deeper call chains. (2) Our current technique cannot handle recursive functions properly.

The amount of `May` covered methods that are actually covered is highly dependent on the type of projects and can range from 6% to 83%. In addition, this number seems to be irrelevant of the types of testing conducted. In order to obtain a more accurate estimate of the code coverage measures using LogCoCo, additional logging statements need to be added into the SUT to reduce the amount of `May` labeled methods. However, the logging locations should be decided strategically (e.g., leveraging techniques like [16, 76]) in order to minimize performance overhead.

5.4 Discussion on the Statement and Branch Coverage

For statement and branch coverage, the accuracy of the `Must-not` labels is 100% for all the experiments. However, the accuracy of the `Must` covered labels ranges from 83% to 100% for statement coverage and 50% to 100% for branch coverage. In seven out of the nine total experiments, the accuracy of the `Must` covered statements is 97% or higher. We manually examined the cases where the LogCoCo results are different from JaCoCo. We summarized them as follows:

1. *limitations on static analysis (LogCoCo issue)*: Java supports polymorphism. The actual type of certain objects are unknown until they are being executed. LogCoCo infers the call graphs statically and mistakenly flags some of the method invocations.
2. *new programming constructs (JaCoCo issue)*: the lambda expression is one of the new programming language constructs introduced in Java 8. JaCoCo mistakenly tags some statements containing lambda expressions as not covered.

The accuracy of the `Must` covered branches is generally above 95%, except one case: during the integration testing of C_3, LogCoCo detected two `Must` covered branches being executed, one of which was falsely labeled. The rationales for the differences of the branch coverage measures are the same as the statement coverage measures.

The amount of `May` actually covered statements and branches are generally higher than the amount of actually covered `May` methods. However, similar to the method-level coverage, we cannot easily guess the actual coverage information for a `May` labeled statement or branch.

5.5 Feedback from the QA Engineers

We demonstrated LogCoCo to the QA engineers at Baidu. They agreed that LogCoCo can be used for their daily testing activities, due to its ease of setup, wider application context, and accurate results. In particular, instead of treating all the source code equally,

Table 2: Comparing the performance of LogCoCo against JaCoCo under various testing activities. The numbers above shows the amount of overlap between LogCoCo and JaCoCo.

Project	Type of Testing	Size of the Logs	Method Coverage (Must		May)	Statement Coverage (Must		May)	Branch Coverage (Must		May)
			(Must	Must-not	May)	(Must	Must-not	May)	(Must	Must-not	May)
C_1	Integration	20 MB	100%	100%	16%	99%	100%	62%	100%	100%	67%
C_2	Unit	600 MB	100%	100%	78%	100%	100%	75%	100%	100%	76%
	Integration	550 MB	100%	100%	30%	100%	100%	90%	100%	100%	83%
C_3	Unit	8 MB	100%	88%	15%	84%	100%	77%	100%	100%	50%
	Integration	26 MB	100%	-	33%	83%	100%	89%	50%	100%	60%
C_4	Integration	29 MB	100%	99%	18%	97%	100%	49%	100%	100%	55%
C_5	Integration	1.1 GB	100%	90%	6%	97%	100%	45%	96%	100%	39%
HBase	Unit	527 MB	100%	83%	83%	99%	100%	83%	100%	100%	76%
	Integration	193 MB	100%	89%	50%	99%	100%	71%	100%	100%	63%

they would pay particular attention to the coverage of the methods, which have logging statements instrumented. This was because many of the logging statements were inserted into risky methods or methods which suffered from past field failures. Having test cases cover these methods is considered a higher priority. LogCoCo addressed this task nicely. In addition, they agreed that LogCoCo can also be used to speed up problem diagnosis in the field by automatically pin-pointing the problematic code regions. Finally, they were also very interested in the amount of `May` labeled entities (a.k.a., methods, statements, and branches), as they knew little about the runtime behavior of these entities. They considered reducing the amount of `May` labeled entities as one approach to improving their existing logging practices and were very interested to collaborate further with us on this topic.

Findings: The accuracy of `Must` and `Must-not` labeled entities from LogCoCo is very high for all three types of code coverage measures. However, one cannot easily infer whether a `May` labeled entity is actually covered in a test.
Implications: To further improve the accuracy, one must reduce the amount of `May` labeled entities through additional instrumentation. Researchers and practitioners can look into existing works (e.g., [16, 76]), which improve the SUT's logging behavior with minimal performance overhead.

6 RQ2: USEFULNESS

Existing code coverage tools are usually applied only during unit or integration testing due to various challenges explained in Section 2. LogCoCo, which analyzes the readily available execution logs, can work on a much wider application context. In this RQ, we intend to check if we can leverage the LogCoCo results from various execution contexts to improve the existing test suites. To tackle this problem, we further split this RQ into the following two sub-RQs:

RQ2.1: Can we improve the in-house functional test suites by the comparison among each other?

In this sub-RQ, we will focus on unit and integration testing, as they have different testing purposes. Unit testing examines the SUT's behavior with respect to the implementation of individual classes, whereas integration testing examines whether individual units can work correctly when they are connected to each other. We intend to check if one can leverage the coverage differences to improve the existing unit or integration test suites using data from LogCoCo.

Experiment. To study this sub-RQ, we reused the data obtained from RQ1's experiments. In particular, we selected the data from two commercial projects: C_2 and C_3, as they contain data from both unit and integration test suites. The main reason we focused on the commercial projects in this sub-RQ was because we can easily get hold of the QA engineers of Baidu for feedback or surveys (e.g., whether they can evaluate and improve the unit or integration tests by comparing the coverage data).

Data Analysis and Discussion. It would be impractical to study all the coverage differences from the two types of tests due to their large size. We randomly sampled a subset of methods, where both types of testing covered but their statement and branch level coverage measures differed. We presented this dataset to QA engineers from the two commercial projects for feedback. After manual examinations, the QA engineers agreed to add additional unit testing cases for all the cases where unit testing did not cover. However, adding additional integration tests is harder than adding additional unit tests. The QA engineers rejected about 85% of the cases where unit testing covered but integration testing missed, as they were considered as hard or lower priority. We summarized their rationales as follows:

- *Defensive Programming*: defensive programming is a programming style to guard against unexpected conditions. Although it generally improves the robustness of the SUT, there can be unnecessary code introduced to guard against errors that would be impossible to happen. Developers insert much error checking code into the systems. Some of these issues are rare or impossible to happen. It is very hard to come up with an integration test case whose input values can exercise certain branches.
- *Low Risk Code*: some of the modules are considered as low risk based on the experience of the QA engineers. Since they are already covered by the unit test suites, adding additional integration test cases is considered as low priority.

RQ2.2: Can we evaluate the representativeness of in-house test suites by comparing them against field behavior?

One of the common concerns associated with QA engineers is whether the existing in-house test suites can properly represent field behavior. We intend to check if one can evaluate the quality of the existing in-house test suites by comparing them against field coverage using data from LogCoCo.

Experiment. Due to confidentiality reasons, we cannot disclose the details about the field behavior for the commercial projects. Therefore, we studied the open source system, HBase, for RQ2.2. The integration test suite from HBase is considered as a comprehensive test suite and is intended for "elaborate proofing of a release candidate beyond what unit tests can do" [22]. Hence, we consider it as HBase's in-house test suite. There are two setup approaches to running the HBase's integration tests: a mini-cluster, or a distributed cluster. The mini-cluster setup is usually run on one machine and can be integrated with the Maven build process. The distributed cluster setup needs a real HBase cluster setup and is invoked using a separated command. In this experiment, we ran under both setups and collected their logs.

The workloads defined in YCSB are derived by examining a wide range of workload characteristics from real web applications [12]. We thus used the YCSB benchmark test suite to mimic the field behavior of HBase. However, as we discovered under default settings, HBase did not output any logs during the benchmarking process. We followed the instructions from [33] to change the log levels for HBase from INFO to DEBUG on the fly (a.k.a., without server reboot). The resulting log file size is around 270 MB when running the YCSB benchmark tests for one hour.

Data Analysis and Discussion. Based on the LogCoCo results, there are 12 methods which were covered by the YCSB test and not by the integration test under the mini-cluster setup. Most of these methods were related to the functionalities associated with cluster setup and communications. Under the distributed cluster setup, which is more realistic, all the covered methods in the YSCB test were covered by the integration test.

The log verbosity level for the unit and the integration tests of HBase in RQ1 was kept as the default INFO level. Both tests generated hundreds of megabytes of logs. However, under the default verbosity level, the YCSB benchmarking test generated no logs except a few lines at the beginning of the test. This is mainly because HBase does not perform logging for their normal read, write, and scan operations for performance concerns. The integration tests output more INFO level logs is because: (1) many of the testing methods are instrumented with INFO or higher level logs. Such logs are not printed in practice; and (2) in addition to the functionalities covered in the YCSB benchmark, integration tests also verify other use cases, which can generate many logs. For example, one integration test case is about region replications. In this test, one of the HBase component, ZooKeeper, which is responsible for distributed configuration and naming service, generated many INFO level logs.

We further assessed the performance impact of turning on the DEBUG level logs for HBase. We compared the response time under the DEBUG and the INFO level logging with YCSB threads configured at 5, 10, and 15, respectively. The performance impact was very small ($< 1\%$) under all three YCSB settings (a.k.a., three different YCSB benchmark runs). Thus, for HBase the impact of DEBUG level logging is much smaller than JaCoCo. Furthermore, compared to JaCoCo, which requires server restart to enable/disable its process, the DEBUG level logging can easily be turned on/off during runtime.

Findings: LogCoCo results can be used to evaluate and improve the existing test suites. Multiple rationales are considered when adding a test case besides coverage.
Implications: There are mature techniques (e.g., EvoSuite [2] and Pex [69]) to automatically generate unit test cases with high coverage. However, there are no such techniques for other types of tests, which still require high manual effort to understand the context and to decide on a case-by-case basis. Further research is needed in this area.
The coverage information from LogCoCo highly depends on amount of generated logs. Researchers or practitioners should look into system monitoring techniques (e.g., sampling [62] or adaptive instrumentation [38]), which maximize the obtained information with minimal logging overhead.

7 RELATED WORK

In this section, we will discuss two areas of related research: (1) code coverage, and (2) software logging.

7.1 Code Coverage

Code coverage measures the amount of source code executed while running SUTs under various scenarios [18]. They have been used widely in both academia and industry to assess and improve the effectiveness of existing test suites [3, 5, 43, 46, 51, 53]. There are quite a few open source (e.g., [11, 32]) and commercial (e.g., [14, 49]) code coverage tools available. All these tools leverage additional code instrumentation, either at the source code level (e.g., [6, 10, 13]) or at the binary/bytecode (e.g., [11, 31, 49]) level, to automatically collect the runtime system behavior in order to measure the code coverage measures. In [21], Häubl et al. derived code coverage information from the profiling data recorded by an the just-in-time (JIT) compiler. They also compared their coverage information against JaCoCo. They showed their results are more accurate than JaCoCo and yield smaller overhead. Häubl et al.'s approach is different from ours, as they relied on data from the underlying virtual machines, whereas we focus on the logging statements from the SUT's source code. Recently, Horváth et al. [27] compared the results from various Java code coverage tools and assessed the impact of their differences to test prioritization and test suite reduction. In this paper, we have evaluated the state-of-the-art Java-based code coverage tools in a field like setting and proposed a new approach, which leverages the readily available execution logs, to automatically estimating the code coverage measures.

In addition to the traditional code coverage metrics (e.g., method, branch, decision, and MC/DC coverage), new metrics have been proposed to better assess the oracle quality [59], to detect untested code regions [28], and to compose test cases with better abilities to detect faults [67]. There are various empirical studies conducted to examine the relationship between the test effectiveness and various code coverage metrics. For example, Inozemtseva and Holmes [30] leveraged mutation testing to evaluate the fault detection effectiveness of various code coverage measures and found that there is a low to moderate correlation between the two. Kochhar et al. [41, 42] performed a similar study, except they used real bugs instead. Their study reported a statistically significant correlation (moderate to strong) between fault detection and code coverage measures. Gligoric et al. [20] compared the effectiveness of various code coverage metrics in the context of test adequacy. The work by Wang et al. [66], which is the closest to our work, compared the coverage between in-house test suites and field executions using invariant-based models. Our work differs from [66] in the following two main areas: (1) they used a record-replay tool, which instruments the SUT to collect coverage measures. Our work estimates the coverage measures based on the existing logs without extra instrumentation. (2) While they mainly focused on the client and desktop-based systems, our focus is on the server-based distributed systems deployed in a field-like environment processing large volumes of concurrent requests. In our context, extra instrumentation would not be ideal, as it will have a negative impact on the user experience.

7.2 Software Logging

Software logging is a cross-cutting concern that scatters across the entire system and inter-mixes with the feature code [39]. Unfortunately, recent empirical studies show that there are no well-established logging practices for commercial [19, 55] and open source systems [8, 74]. Recently researchers have focused on providing automated logging suggestions based on learning from past logging practices [9, 19, 37, 77] or program analysis [76]. Execution logs are widely available inside large-scale software systems for anomaly detection [26, 61], system monitoring [52, 60], problem debugging [7, 44, 73, 75], test analysis [35, 63], and business decision making [4]. Our work was inspired by [75], which leveraged logs to infer executed code paths for problem diagnosis. However, this is the first work, to the authors' knowledge, which uses logs to automatically estimate code coverage measures.

The limitation is that we rely on that the study systems contain sufficient logging. It is generally the case among server-side projects. For client-side or other projects with limited logging, our approach should be complementary by other code coverage tools.

8 THREATS TO VALIDITY

In this section, we will discuss the threats to validity.

8.1 Internal Validity

In this paper, we proposed an automated approach to estimating code coverage by analyzing the readily available execution logs. The performance of our approach highly depends on the amount of the logging and the verbosity levels. The amount of logging is not a major issue as existing empirical studies show that software logging is pervasive in both open source [8, 74] and commercial [19, 55] systems. The logging overhead due to lower verbosity levels is small for the HBase study. For other systems, one can choose to enable lower verbosity level for a short period of time or use advanced logging techniques like sampling and adaptive instrumentation.

8.2 External Validity

In this paper, we focused on the server-side systems mainly because these systems use logs extensively for a variety of tasks. All these systems are under active development and used by millions of users worldwide. To ensure our approach is generic, we studied both commercial and open source systems. Although our approach was evaluated on Java systems, to support other programming languages, we just need to replace the parser for another language (e.g., Saturn [70] for C, and AST [56] for Python) in LogCoCo. The remaining process stays the same. Our findings in the case studies may not be generalizable to systems and tools which have no or very few logging statements (sometimes seen in mobile applications and client/desktop-based systems).

8.3 Construct Validity

When comparing the results between LogCoCo and the state-of-the-art code coverage tools, we focused on JaCoCo, which collects code coverage information via bytecode instrumentation. This is because: (1) JaCoCo is widely used in Baidu, so we can easily collect the code coverage for the systems and gather feedback from the QA engineers; and (2) we intend to assess the coverage measures in a field-like environment, in which the system is deployed in a distributed environment and used by millions of users. Source-code-instrumentation-based code coverage tools (e.g., [10]) are not ideal, as they require recompilation and redeployment of the SUT.

9 CONCLUSIONS AND FUTURE WORK

Existing code coverage tools suffer from various problems, which limit their application context. To overcome with these problems, this paper presents a novel approach, LogCoCo, which automatically estimates the code coverage measures by using the readily available execution logs. We have evaluated LogCoCo on a variety of testing activities conducted on open source and commercial systems. Our results show that LogCoCo yields high accuracy and can be used to evaluate and improve existing test suites.

In the future, we plan to extend LogCoCo for other programming languages. In particular, we are interested in applying LogCoCo to systems implemented in multiple programming languages. Furthermore, we also intend to extend LogCoCo to support other coverage criteria (e.g., data-flow coverage and concurrency coverage). Finally, since the quality of the LogCoCo results highly depends on the quality of logging, we will research into cost-effective techniques to improve the existing logging code.

ACKNOWLEDGEMENT

This work is done during the first author's internship at Baidu. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Baidu and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of Baidu's products.

REFERENCES

[1] Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. 2011. Code Coverage Analysis in Practice for Large Systems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*.

[2] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults

in a Financial Application. In *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE SEIP)*.

[3] Paul Ammann and Jeff Offutt. 2017. *Introduction to Software Testing* (2nd ed.). Cambridge University Press, New York, NY, USA.

[4] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The Bones of the System: A Case Study of Logging and Telemetry at Microsoft. In *Companion Proceedings of the 38th International Conference on Software Engineering)*.

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* (2015).

[6] Ira Baxter. 2002. Branch Coverage for Arbitrary Languages Made Easy. http://www.semdesigns.com/Company/Publications/TestCoverage.pdf. Last accessed: 01/29/2018.

[7] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (FSE)*.

[8] Boyuan Chen and Zhen Ming (Jack) Jiang. 2016. Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation. *Empirical Software Engineering* (2016).

[9] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-patterns in the Logging Code. In *Proceedings of the 39th International Conference on Software Engineering*.

[10] Clover. 2018. Java and Groovy code coverage. https://www.atlassian.com/software/clover. Last accessed: 03/04/2018.

[11] Cobertuna. 2018. Cobertura. http://cobertura.github.io/cobertura/. Last accessed: 01/29/2018.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*.

[13] Coverage.py. 2018. A tool for measuring code coverage of Python programs. https://coverage.readthedocs.io/en/coverage-4.5.1/. Last accessed: 01/29/2018.

[14] Semantic Designs. 2016. Test Coverage tools. http://www.semdesigns.com/Products/TestCoverage/. Last accessed: 01/29/2018.

[15] Edsger W. Dijkstra. 1970. Notes on Structured Programming. (April 1970).

[16] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-aware Logging Mechanism for Performance Diagnosis. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC)*.

[17] Eclipse. 2018. JDT Java Development Tools. https://eclipse.org/jdt/. Last accessed: 08/26/2016.

[18] Martin Fowler. 2012. TestCoverage. https://martinfowler.com/bliki/TestCoverage.html. Last accessed: 01/29/2018.

[19] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*.

[20] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-adequate Test Suites Using Coverage Criteria. In *In Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*.

[21] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2013. Deriving Code Coverage Information from Profiling Data Recorded for a Trace-based Just-in-time Compiler. In *In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*.

[22] HBase. 2012. Integration Tests for HBase. http://hbase.apache.org/0.94/book/hbase.tests.html. Last accessed: 01/29/2018.

[23] HBase. 2018. Apache HBase. https://hbase.apache.org. Last accessed: 01/29/2018.

[24] HBase. 2018. Powered By Apache HBase. http://hbase.apache.org/poweredbyhbase.html. Last accessed: 03/04/2018.

[25] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2017. Towards Automated Log Parsing for Large-Scale Log Data Analysis. *IEEE Transactions on Dependable and Secure Computing* (2017).

[26] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2016. Experience Report: System Log Analysis for Anomaly Detection. In *In Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*.

[27] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. 2017. Code coverage differences of Java bytecode and source code instrumentation tools. *Software Quality Journal* (Dec 2017).

[28] Chen Huo and James Clause. 2016. Interpreting Coverage Information Using Direct and Indirect Coverage. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[29] IBM. 2017. IBM Rational Test Realtie - Estimating Instrumentation Overhead. https://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/tsciestimate.htm. Last accessed: 01/29/2018.

[30] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*.

[31] Jacoco. 2018. JaCoCo Implementation Design. http://www.jacoco.org/jacoco/trunk/doc/implementation.html. Last accessed: 01/29/2018.

[32] JaCoCo. 2018. JaCoCo Java Code Coverage library. http://www.eclemma.org/jacoco/. Last accessed: 01/29/2018.

[33] Vincent Jiang. 2016. How to change HBase log level on the fly? https://community.hortonworks.com/content/supportkb/49499/how-to-change-hbase-log-level-on-the-fly.html. Last accessed: 01/29/2018.

[34] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. An Automated Approach for Abstracting Execution Logs to Execution Events. *Journal of Software Maintaince and Evolution* 20, 4 (July 2008).

[35] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated Performance Analysis of Load Tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*.

[36] Paul C. Jorgensen. 2008. *Software Testing: A Craftsman's Approach* (3rd ed.). Auerbach Publications, Boston, MA, USA.

[37] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging Library Migrations: A Case Study for the Apache Software Foundation Projects. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*.

[38] Emre Kiciman and Helen J. Wang. 2007. Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*.

[39] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. *Aspect-oriented programming*.

[40] Gene Kim, Patrick Debois, John Willis, and Jez Humble. 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*.

[41] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. [n. d.]. PCode Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability* ([n. d.]).

[42] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

[43] Pavneet Singh Kochhar, Ferdian Thungand, David Lo, and Julia Lawall. 2014. An Empirical Study on the Adequacy of Testing in Open Source Projects. In *2014 21st Asia-Pacific Software Engineering Conference (APSEC)*.

[44] Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, and Dongmei Zhang. 2016. iDice: Problem Identification for Emerging Issues. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*.

[45] Log4J. 2012. LOG4J A logging library for Java. http://logging.apache.org/log4j/1.2/. Last accessed: 07/17/2018.

[46] Leonardo Mariani, Dan Hao, Rajesh Subramanyan, and Hong Zhu. 2017. The central role of test automation in software quality assurance. *Software Quality Journal* (2017).

[47] Google Zúrich Marko Ivanković. 2014. Measuring Coverage at Google. https://testing.googleblog.com/2014/07/measuring-coverage-at-google.html. Last accessed: 01/29/2018.

[48] Scott Matteson. 2018. https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/.

[49] Microsoft. 2016. Microsoft Visual Studio - Using Code Coverage to Determine How Much Code is being Tested. https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested. Last accessed: 01/29/2018.

[50] Kannan Muthukkaruppan. 2010. The Underlying Technology of Messages. https://www.facebook.com/note.php?note_id=454991608919. Last accessed: 03/04/2018.

[51] Hoan Anh Nguyen, Tung Thanh Nguyen, Tung Thanh Nguyen, and Hung Viet Nguyen. 2017. Interaction-Based Tracking of Program Entities for Test Case Evolution. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[52] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Communications of ACM* (2012).

[53] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering (FOSE)*.

[54] Alan Page and Ken Johnston. 2008. *How We Test Software at Microsoft*. Microsoft Press.

[55] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. In *Companion Proceedings of the 37th International Conference on Software Engineering*.

[56] Python. 2018. The AST module in Python standard library. https://docs.python.org/2/library/ast.html. Last accessed: 04/02/2018.

[57] Ajitha Rajan, Michael Whalen, and Mats Heimdahl. 2008. The effect of program and model structure on mc/dc test adequacy coverage. In *In Proceedings of the*

ACM/IEEE 30th International Conference on Software Engineering (ICSE).

[58] Alberto Savoia. 2010. Code coverage goal: 80% and no less! https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html. Last accessed: 01/29/2018.

[59] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *In Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST).*

[60] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. 2014. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process* (2014).

[61] Weiyi Shang, Zhen Ming (Jack) Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. 2013. Assisting Developers of Big Data Analytics Applications when Deploying on Hadoop Clouds. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE).*

[62] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[63] Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E. Hassan. 2017. Continuous validation of performance test workloads. *Automated Software Engineering* (2017).

[64] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2002. Efficient Instrumentation for Code Coverage Testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).*

[65] Tumblr. 2018. Tumblr Architecture - 15 Billion Page Views A Month And Harder To Scale Than Twitter. http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html. Last accessed: 03/04/2018.

[66] Qianqian Wang, Yuriy Brun, and Alessandro Orso. 2017. Behavioral Execution Comparison: Are Tests Representative of Field Behavior?. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).*

[67] Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. 2013. Observable Modified Condition/Decision Coverage. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE).*

[68] Michael Würsch, Emanuel Giger, and Harald C. Gall. 2013. Evaluating a Query Framework for Software Evolution Data. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4 (October 2013).

[69] Tao Xie, Nikolai Tillmann, and Pratap Lakshman. 2016. Advances in Unit Testing: Theory and Practice. In *Proceedings of the 38th International Conference on Software Engineering Companion.*

[70] Yichen Xie and Alex Aiken. 2007. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM Trans. Program. Lang. Syst.* (May 2007).

[71] XWiki. 2018. XWiki Development Zone - Testing. http://dev.xwiki.org/xwiki/bin/view/Community/Testing. Last accessed: 03/04/2018.

[72] Yahoo. 2018. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/. Last accessed: 01/29/2018.

[73] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI).*

[74] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-source Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).*

[75] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[76] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles.*

[77] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proceedings of the 37th International Conference on Software Engineering.*

[78] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering.*