Efficient Incremental Code Coverage Analysis for Regression Test Suites

Jiale Amber Wang*
University of Waterloo
Canada
jiale.wang@uwaterloo.ca

Kaiyuan Wang*
Google
USA
kaiyuanw@google.com

Pengyu Nie
University of Waterloo
Canada
pynie@uwaterloo.ca

Abstract

Code coverage analysis has been widely adopted in the continuous integration of open-source and industry software repositories to monitor the adequacy of regression test suites. However, computing code coverage can be costly, introducing significant overhead during test execution. Plus, re-collecting code coverage for the entire test suite is usually unnecessary when only a part of the coverage data is affected by code changes. While regression test selection (RTS) techniques exist to select a subset of tests whose behaviors may be affected by code changes, they are not compatible with code coverage analysis techniques—that is, simply executing RTS-selected tests leads to incorrect code coverage results.

In this paper, we present the first incremental code coverage analysis technique, which speeds up code coverage analysis by executing a minimal subset of tests to update the coverage data affected by code changes. We implement our technique in a tool dubbed ijaCoCo, which builds on Ekstazi and JaCoCo—the state-of-the-art RTS and code coverage analysis tools for Java. We evaluate ijaCoCo on 1,122 versions from 22 open-source repositories and show that ijaCoCo can speed up code coverage analysis time by an average of 1.86× and up to 8.20× compared to JaCoCo.

CCS Concepts

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

Keywords

regression testing, code coverage analysis, regression test selection

*Both authors contributed equally to this research.

1 Introduction

Developers rely on high quality tests to assess the correctness of software systems. By setting up testing in continuous integration (CI), i.e., executing test suites on every version, developers can quickly detect regressions [12, 29]. *Code coverage* [20, 24, 28, 32, 53, 64, 71] is a well-established measurement for the test suite adequacy. It is defined as the portion of code elements (e.g., instructions, lines, branches) that are transitively executed during the execution of a test suite [48].

Code coverage analysis is widely adopted in real-world software development process. For example, JaCoCo [61], one of the most popular code coverage analysis tools for Java, is used by more than 395K open-source repositories on GitHub [21]. Large industry companies such as Google and IBM also see the importance of setting up code coverage analysis as a part of their internal CI pipelines [1, 4, 30, 31].

However, code coverage analysis can be time-consuming. A typical code coverage analysis technique needs to instrument the codebase to insert probes into code elements, and then execute the test suite to collect *coverage data*—recording which probes (and the code elements between them) are covered. Obviously, executing the test suites on each version is already costly and time-intensive [47, 63]. Performing code coverage analysis on top of it adds more overhead [1, 30, 31]. This additional overhead adds up over time as the codebase evolves and scales.

To speed up regression testing during software evolution, researchers have proposed *regression test selection* (RTS) [17, 18, 23, 25, 36, 65], which only executes selected tests whose behaviors may be affected by code changes. Executing the other tests that depend solely on unchanged code is unnecessary, because their behaviors should not change. Similarly, during software evolution, performing code coverage for the entire codebase by executing all tests on each version is not only time-consuming but also unnecessary. The code coverage for the part of the codebase that solely depend on unchanged code should not change either.

In this paper, we propose an efficient *incremental code coverage analysis* technique to collect code coverage for regression test suites. Given two software versions, the coverage data collected on the old version, and the code changes, incremental code coverage analysis selects a minimal set of tests that need to be executed to update the coverage data for the new version. Surprisingly, naively running RTS and then running code coverage analysis with RTS-selected tests may not be efficient or correct. The reason is that collecting coverage data at the per-test granularity has performance and safety issues. Thus, we propose to expand the set of RTS-selected tests in a way to ensure that the two versions' coverage data can be correctly merged (a detailed explanation with example is given in §4.2).

To demonstrate the effectiveness of incremental code coverage analysis, we implement ɪJaCoCo, a tool that **i**ncrementally collects **Ja**va source **co**de **co**verage. ɪJaCoCo it built on top of JaCoCo [61], a widely-used code coverage analysis tool for Java, and Ekstazi [22, 23], a state-of-the-art file-level dynamic RTS tool for Java. ɪJaCoCo is designed to have the same interface as JaCoCo (i.e., can be used as a command line tool and a Maven plugin), such that existing users of JaCoCo can seamlessly switch to ɪJaCoCo. We envision two usage scenarios of ɪJaCoCo: (1) speeding up code coverage analysis for each code version on CI; (2) instantly providing code coverage feedback when developers edit code on local machines.

We evaluated ɪJaCoCo on a dataset of 1,122 versions from 22 open-source repositories, with 1.1M lines of code in total. Compared with JaCoCo, ɪJaCoCo can achieve an end-to-end time speedup of 1.86× on average and up to 8.20×. While ɪJaCoCo incurs overhead on the first version of each repository to build the dependency graph, in most subsequent versions, ɪJaCoCo only needs to run a small subset of tests to update coverage data. Thus, incremental code coverage analysis allows for lower CI cost and fast feedback during software development. We also compared the test selection rate of ɪJaCoCo with Ekstazi, and found that although ɪJaCoCo needs to select about twice as many tests as Ekstazi to correctly compute coverage data, the speedup of ɪJaCoCo is still significant.

The *correctness* of incremental code coverage analysis is crucial, i.e., the coverage data collected incrementally should match the coverage data collected from running all tests with the existing code coverage analysis technique. We prove that our incremental code coverage analysis technique is correct as long as the underlying RTS technique is *safe*. An RTS technique is safe if it does not miss any test whose behavior may be affected by a code change. ɪJaCoCo is built on top of Ekstazi, which is a safe RTS tool for Java repositories [70]. Moreover, we confirmed that the code coverage measured by ɪJaCoCo and JaCoCo are consistent across all versions in our experiment.

The main contributions of this work include:

- **Idea**. We demonstrate that the idea of incremental computation can speed up code coverage analysis for regression test suites.

- **Technique**. We design the first incremental code coverage analysis technique that integrates code coverage analysis and RTS techniques; note that a non-trivial integration is needed to correctly update the coverage data when executing a subset of tests.

- **Implementation**. We implement our technique as ɪJaCoCo, an industrial-level incremental code coverage analysis tool for Java.

- **Evaluation**. Our evaluation found that ɪJaCoCo can speed up code coverage analysis end-to-end time by 1.86× on average and up to 8.20× compared to JaCoCo.

The replication package of ɪJaCoCo, including the tool, our experiment scripts, and results, is open-sourced at: https://github.com/uw-swag/ijacoco

2 Motivating Example

Figure 1 shows a real-world example from the `commons-lang` repository[1] where incremental code coverage analysis is helpful. This

[1]https://github.com/apache/commons-lang.git

```
// org/apache/commons/lang3/reflect/FieldUtils.java
public static void removeFinalModifier(final Field field,
    final boolean forceAccess) {
  Validate.isTrue(field != null, "The field must not be null");
  try { ...
- } catch (final NoSuchFieldException ignored) {
+ } catch (final NoSuchFieldException | IllegalAccessException ignored) {
    // The field class contains always a modifiers field
- } catch (final IllegalAccessException ignored) {
-   // The modifiers field is made accessible
  }
}
```

(a) Code change between version `b1deb442` and `9fb4f47f`.

```
org.apache.commons.lang3.builder.ReflectionDiffBuilderTest
org.apache.commons.lang3.reflect.FieldUtilsTest
org.apache.commons.lang3.reflect.TypeUtilsTest
org.apache.commons.lang3.time.StopWatchTest
```

(b) Tests selected by ɪJaCoCo.

Figure 1: Example of using ɪJaCoCo on `commons-lang`: when computing the code coverage on version `9fb4f47f`, JaCoCo executes all 149 tests and takes 29.75s, and ɪJaCoCo only executes 4 tests and takes 18.65s (speedup: 1.60×).

repository is configured to run JaCoCo to collect code coverage on every new version on GitHub's CI. On version `9fb4f47f`, JaCoCo takes 29.75s to execute all 149 tests and collect code coverage. However, the code change between version `9fb4f47f` and the previous version `b1deb442` is rather small: as shown in Figure 1a, the only change is merging two `catch` blocks into one. Re-computing code coverage for the entire codebase is unnecessary because only a small part of the codebase is affected by the change.

Our proposed incremental code coverage analysis technique first analyzes the code change, finds the part of the codebase whose code coverage may be affected by the change, and computes the tests need to be executed to collect the coverage data for the affected part. When applying ɪJaCoCo on this example, it finds out that only 4 tests need to be executed, as shown in Figure 1b. The end-to-end time for ɪJaCoCo to analyze code changes, execute the selected tests, and collect code coverage is 18.65s, which is 1.60 times faster than JaCoCo.

3 Background

In this section, we briefly introduce the code coverage analysis (§3.1) and RTS (§3.2) techniques, which are the basis of our work.

3.1 Code Coverage Analysis

Code coverage [4, 10, 15, 20, 24, 28, 30, 32, 53, 64, 71] measures the adequacy of a test suite by quantifying the portion of code elements (e.g., lines) that have been covered (i.e., transitively executed) by the tests. The inputs to code coverage analysis include a codebase with a test suite, and the output is coverage data, denoted as $\mathcal{D}$, which records whether each code element is covered. Code coverage analysis usually generates a human-readable report based on $\mathcal{D}$, including code coverage percentage at various granularities (e.g., line coverage, branch coverage), and highlights the uncovered code elements.

A typical code coverage analysis tool, such as JaCoCo [61], maintains coverage data in the format of a mapping from source code class (or file in non-object oriented programming languages) to a set of *probes* (denoted as $\mathcal{P} = \{p\}$) that are inserted into the class and then executed: $\mathcal{D} = \{c \mapsto \mathcal{P}\}$. A probe p is an additional instruction instrumented by the tool at the beginning of each basic block, which upon execution, adds itself to $\mathcal{D}[p.\text{class}]$. Note that it is not necessary to place a probe before every code element; instead, multiple code elements in the same basic block (i.e., on the same execution path) can share one probe. If a probe is executed during tests, all the code elements in that basic block are considered as covered. A common code coverage analysis technique first instruments the codebase to insert probes, then executes the tests to collect the coverage data $\mathcal{D}$, and finally generates a report based on $\mathcal{D}$.

3.2 Regression Test Selection (RTS)

Regression test selection [22, 23, 36, 37, 42, 68] speeds up regression testing by only executing tests that are affected by code changes. The inputs to RTS include two versions of a codebase, the test suite on the new version, and dependency graph from the old version. The outputs are a subset of the test suite whose behavior may change due to the code changes, and an updated dependency graph to be used in the next version. The workflow of an RTS tool has three phases: the analysis phase selects tests based on code changes and the last version's dependency graph; the execution phase executes the selected tests; and the collection phase collects the updated dependency graph. RTS techniques vary by the granularity of the dependency graph and whether it is collected statically or dynamically. Prior work finds that using class-level (or file-level) dependency graph is a "sweet spot" with low analysis overhead and decent test selection capability [22, 23, 36, 37]; using more fine-grained dependency graph (e.g., method-level) helps RTS to be more precise (i.e., avoid selecting tests not affected by code changes) but usually at the cost of higher analysis overhead and more engineering effort [42, 68]. Dynamic RTS (i.e., collecting dependency graph via dynamic analysis) is usually more precise and safer; static RTS (i.e., collecting dependency graph via static analysis) can be faster and easier to perform offline (isolated from the execution phase) [36].

In this work, we adopt the class-level dynamic RTS technique because its good overall precision and safety. We describe its three phases in more details. Given two software versions, let C be the classes and $\mathcal{T}$ be the tests[2] on the old version; let C' and $\mathcal{T}'$ be the classes and tests on the new version. RTS requires the dependency graph collected from the old version $\mathcal{G} = \{t \mapsto c\}$, where: $t \in \mathcal{T}$ is a test, c is a (test or non-test) class that t *transitively* depends on; by definition, $(t \mapsto t) \in \mathcal{G}$.

The goal of the analysis phase is to select a subset of tests to be executed: $\mathcal{T}'_{\text{rts}} \subseteq \mathcal{T}'$. In this phase, RTS first figures out the set of classes that have changed $C_\Delta \subseteq C$, and then selects the tests that (1) depend on a changed class; or (2) are added in the new version: $\mathcal{T}'_{\text{rts}} = \{t \in \mathcal{T}' \mid (t \mapsto c) \in \mathcal{G} \wedge c \in C_\Delta\} \cup \mathcal{T}' \setminus \mathcal{T}$. Specially, on the first version of using RTS, all test classes are selected ($\mathcal{T}'_{\text{rts}} = \mathcal{T}'$).

The execution phase, which executes the selected test classes, and the collection phase, which updates the dependency graph, are
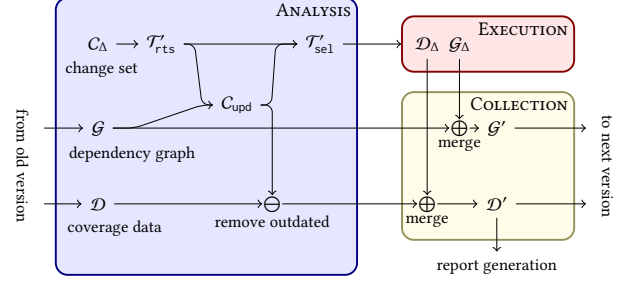
---
[2]We use "tests" to refer to test classes in this paper.

Figure 2: Workflow of incremental code coverage analysis.

usually tightly integrated. Specifically, RTS performs instrumentation before executing tests to insert instructions that record the dependency between (selected) tests and classes: $\mathcal{G}_\Delta = \{t \mapsto c\}$ for $t \in \mathcal{T}'_{\text{rts}}$. Then, the updated dependency graph can be represented as $\mathcal{G}' = \{(t \mapsto c) \in \mathcal{G} \mid t \notin \mathcal{T}'_{\text{rts}}\} \cup \mathcal{G}_\Delta$. The updated dependency graph will be used by RTS in the next version.

4 Incremental Code Coverage Analysis

Figure 2 shows the workflow of incremental code coverage analysis, consisting of three phases: analysis, execution, and collection. We first define the inputs and outputs of the workflow (§4.1), then describe the three phases (§4.2–§4.4), and finally prove the correctness of our technique (§4.5). We use symbols without prime (e.g., $\mathcal{G}$) to denote the data on the old version, and symbols with prime (e.g., $\mathcal{G}'$) to denote the data on the new version.

4.1 Inputs and Outputs

Incremental code coverage analysis requires four inputs: the old and new versions of the codebase, the dependency graph collected from the old version $\mathcal{G}$ (similar to RTS), and the coverage data collected from the old version $\mathcal{D}$. In this work, we focus on integrating class-level dynamic RTS and code coverage analysis which groups coverage data at class-level. Based on the findings in related work on RTS [22, 23, 36, 37] and the fact that existing code coverage analysis tools group coverage data at class-level (such as JaCoCo [61]), we believe that integrating the two techniques at class-level would lead to the best performance; we leave the exploration of other levels of integration to future work. Specifically, the changeset $C_\Delta = \{c\}$ is the set of classes that changed; the dependency graph $\mathcal{G} = \{t \mapsto c\}$ maps each test to the (test or non-test) classes it *transitively* depends on; and the coverage data $\mathcal{D} = \{c \mapsto \mathcal{P}\}$ maps each class to the probes in that class that are executed during testing.

The outputs of incremental code coverage analysis include: (1) the updated dependency graph $\mathcal{G}'$, which will be used by the analysis phase in the version; (2) the coverage data $\mathcal{D}'$, which is used to generate coverage reports and will also be used by the next version.

4.2 Analysis Phase

The goal of the analysis phase is to select a subset of tests that should be executed. We start from the set of tests selected by RTS,
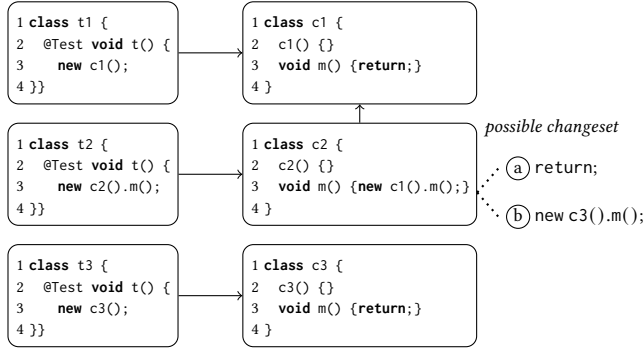
Figure 3: Example showing the necessity for incremental code coverage analysis selecting more test classes than RTS; directed edges represent "depends on" relationship.

denoted by $\mathcal{T}'_{\text{rts}}$:

$$\mathcal{T}'_{\text{rts}} = \text{RTS}(\mathcal{G}, C_\Delta)$$
$$= \{t \in \mathcal{T}' \mid (t \mapsto c) \in \mathcal{G} \wedge c \in C_\Delta\} \cup \mathcal{T}' \setminus \mathcal{T}$$

These tests must be executed because their behaviors may change due to the changeset.

However, for the purpose of updating code coverage, executing only RTS-selected tests may be insufficient. Let's first consider the part of coverage data that should be updated after the changeset. Since the tests in $\mathcal{T}'_{\text{rts}}$ must be executed, the coverage data for all the classes that they transitively depend on may change; we denote this set of classes as C_{upd} which can be computed by looking up $\mathcal{G}$:

$$C_{\text{upd}} = \bigcup_{t \in \mathcal{T}'_{\text{rts}}} \mathcal{G}[t]$$

The old coverage data for the classes in C_{upd} should be discarded, because the tests in $\mathcal{T}'_{\text{rts}}$ may execute in different paths in the new version and cover different code elements than before. To collect the new coverage data for the classes in C_{upd}, we need to execute all tests that depend on them, denoted as $\mathcal{T}'_{\text{sel}}$ (which is also the final set of selected tests):

$$\mathcal{T}'_{\text{sel}} = \{t \mid (t \mapsto c) \in \mathcal{G} \wedge c \in C_{\text{upd}}\}$$

Note that $\mathcal{T}'_{\text{rts}} \subseteq \mathcal{T}'_{\text{sel}}$, because for any test in $\mathcal{T}'_{\text{rts}}$, there must be some classes in C_{upd} that it depends on.

Necessity for selecting more tests than RTS. One may wonder why we need to select additional tests in $\mathcal{T}'_{\text{sel}}$ compared to $\mathcal{T}'_{\text{rts}}$. Figure 3 provides a counter-example where selecting only $\mathcal{T}'_{\text{rts}}$ is not sufficient. In this example, t_1, t_2 are tests, and c_1, c_2, c_3 are non-test classes. The dependency graph on the old version is:

$$\mathcal{G} = \begin{Bmatrix} t_1 \mapsto c_1 & t_2 \mapsto c_1 & t_3 \mapsto c_3 \\ t_1 \mapsto t_1 & t_2 \mapsto c_2 & t_3 \mapsto t_3 \\ & t_2 \mapsto t_2 & \end{Bmatrix}$$

On the shown (old) version, the covered lines include c_1's lines 2–3, c_2's lines 2–3, and c_3's line 2. Consider the changeset (a) which modifies c_2's line 3 to "return;", i.e., removing its dependency to c_1. RTS will only select $\mathcal{T}'_{\text{rts}} = \{t_2\}$. Executing t_2 will still cover c_2's lines, but no longer covers c_1's lines. In fact, no test would

cover c_1's line 3 after the change. Since the change may results in a decrease in c_1's coverage, without executing t_1 or knowing which lines of c_1 are covered by t_1 and t_2, we cannot update the coverage data for c_1 correctly. Thus, the final selected tests should be $\mathcal{T}'_{\text{sel}} = \{t_1, t_2\}$.

An alternative approach, which has been explored in a prior work on using code coverage information to assist regression testing selection [11], is to collect coverage data of each test separately. However, this approach is inefficient because it incurs overhead of (1) storing multiple copies of coverage data; and (2) merging the coverage data from all tests to compute the union of the sets of probes covered, increasing the time complexity from $O(|C|)$ to $O(|\mathcal{T}||C|)$; similar observations have been reported in prior work [49]. When making this design decision, we performed a preliminary study on our dataset and found that performing code coverage analysis for each test separately (i.e., storing one coverage data per test but without merging them yet) causes an average of 76% overhead when compared to performing code coverage analysis once for the entire test suite.

Moreover, collecting coverage data per test may also result in missing coverage data of some tests, because certain code elements (such as static initializers) are only executed once across the entire test suite, and thus will only be recorded by the first test that executes them. Carefully handling such cases will require engineering effort and may incur more overhead. As a consequence, we follow the common design decision of code coverage analysis tools to collect one copy of coverage data for the entire test suite.

4.3 Execution Phase

The execution phase executes the selected tests $\mathcal{T}'_{\text{sel}}$, with instrumentation required by both code coverage analysis and RTS. For code coverage analysis, we insert probes to record which code elements are executed during execution; at the end of the execution, we should get the incremental coverage data $\mathcal{D}_\Delta$ for the selected tests. For RTS, we insert instructions to record all the classes that each test transitively depends on during execution; at the end, we should get the incremental dependency graph $\mathcal{G}_\Delta$ for the selected tests. The two sets of instrumentation do not interfere with each other because RTS's instrumentation never change the program execution path (e.g., it never adds, removes, or updates branches).

4.4 Collection Phase

The collection phase updates the dependency graph and merges the coverage data. Namely, to be able to perform RTS in the next version, an updated dependency graph is computed based on the information collected from the execution phase:

$$\mathcal{G}' = \{(t \mapsto c) \in \mathcal{G} \mid t \notin \mathcal{T}'_{\text{sel}}\} \cup \mathcal{G}_\Delta$$

Although comparing to RTS (§3.2), we update the dependency graph of more tests (recall that $\mathcal{T}'_{\text{rts}} \subseteq \mathcal{T}'_{\text{sel}}$), it will not affect the correctness of the updated dependency graph.

Then, we update the coverage data by integrating the incremental coverage data $\mathcal{D}_\Delta$ collected during the execution phase into the old coverage data $\mathcal{D}$:

$$\mathcal{D}' = \{(c \mapsto e) \in \mathcal{D} \mid c \notin C_{\text{upd}}\} \cup \mathcal{D}_\Delta$$

Note that the coverage data for the classes in C_{upd} are completely overwritten, and the coverage data for the other classes are *merged*. To illustrate the merge process, consider the changeset (b) in Figure 3 which modifies c_1's line 2 to "new c3().m();". Recall that the covered lines for c_3 in the old version is line 2, and the selected tests $\mathcal{T}'_{sel} = \{t_1, t_2\}$. When executing t_2, line 3 of c_3 is also covered. Since there is no test that depends on c_3 whose execution paths may change due to the changeset, there will not be any reduction in c_3's coverage. Therefore, the old coverage data for c_3 should be kept and merged with the incremental coverage data collected during the execution phase. In this case, $\mathcal{D} = \{c_3 \mapsto \{2\}, ...\}$, $\mathcal{D}_\Delta = \{c_3 \mapsto \{3\}, ...\}$, and thus $\mathcal{D}' = \{c_3 \mapsto \{2, 3\}, ...\}$. This is consistent with the result of re-executing all tests in the new version.

4.5 Correctness

In this subsection, we prove the correctness of incremental code coverage analysis. Our baseline is traditional code coverage analysis (§3.1), which collects code coverage by executing all tests. Correctness in our case means that the coverage data collected by incremental code coverage analysis is the same as the one collected by the baseline. The correctness of our technique depends on the safety of the RTS technique. A safe RTS selects all tests whose behaviors might be affected by the changeset.

Theorem. Incremental code coverage analysis is correct if the underlying RTS technique is safe.

Proof. Given the changeset C_Δ, a safe RTS technique should select all tests $\mathcal{T}'_{rts}$ whose behaviors may change. This means that in the extreme case, all classes in C_{upd} (recall this is the set of classes any test in $\mathcal{T}'_{rts}$ transitively depends on) may have their coverage increased or decreased. That is why we need to execute all tests in $\mathcal{T}'_{sel}$ to completely overwrite the coverage data for classes in C_{upd}.

Now assume there is a class $c \notin C_{upd}$ that is a transitive dependent of some tests in $\mathcal{T}'_{sel}$. We assert that coverage data for c can only increase due to C_Δ (and thus is safe to be merged with $\mathcal{D}$ from the old version). This is because if the changeset result in any decrease in c's coverage, then changeset depends on c, which indicates that $c \in C_{upd}$. This contradicts with our assumption. □

5 IJACOCO Implementation

IJACOCO is built on top of JaCoCo and Ekstazi as the underlying code coverage analysis and RTS tools, respectively. In this section, we first describe how each phase of IJACOCO is implemented in §5.1–§5.3, but focus on integration details rather than repeating the techniques already described in §4. Then, §5.4 describes the usage of IJACOCO as a plugin to the Maven build system.

5.1 Analysis Phase

The analysis phase starts with performing RTS. IJACOCO makes no change to Ekstazi's dependency graph format, which is a list of checksums of all dependent classes for each test. When computing the checksum of a class, Ekstazi would remove all debugging information from the class file such that code changes that do not affect test execution are ignored (e.g., renaming variables or updating comments). However, since JaCoCo identifies probes by line numbers, it is important to include the line number table into the checksum computation. Once Ekstazi has selected the tests $\mathcal{T}'_{rts}$,

IJACOCO follows the steps in §4.2 to compute the set of tests ($\mathcal{T}'_{sel}$) that need to be executed.

5.2 Execution Phase

Both Ekstazi and JaCoCo needs to instrument the codebase at the time of test execution using the javaagent mechanism [3]. As described in §4.3, the two sets of instrumentation do not interfere with each other and thus their ordering does not matter.

5.3 Collection Phase

The collection phase is where the dependency graph is updated and coverage data is collected. IJACOCO also makes no change to JaCoCo's coverage data format (i.e., a hash map stored as the jacoco.exec file). To correctly compute the coverage data, IJACOCO first loads the old version's coverage data from the file system (if available), removes the entries that belongs to the classes (C_{upd}) whose coverage may be affected, and then unions it with the collected coverage data diff ($\mathcal{D}_\Delta$) during execution.

Coverage report generation. Since the format of coverage data is unchanged, IJACOCO reuses JaCoCo's report generation functionalities. Specifically, JaCoCo's report generation takes as inputs the coverage data $\mathcal{D}'$ and the codebase. The codebase is required to recover the mapping between probes and code elements, such that the set of executed probes can be translated into the set of code elements covered, denoted as $\mathcal{E}'_{cov}$. Code coverage metrics can be computed as the percentage of covered code elements out of all code elements: $cov' = \frac{|\mathcal{E}'_{cov}|}{|\mathcal{E}|}$. JaCoCo computes code coverage at line-level, instruction-level, branch-level, and method-level. The report also includes the code coverage metrics over the entire repository, or within a given package or class. Moreover, for each source code file, the report also includes a visualization (in HTML format) to annotate which lines and branches are covered.

5.4 Maven Plugin

IJACOCO is shipped with a Maven plugin, just like Ekstazi and JaCoCo, so that it can be simply integrated into repositories using the Maven build system. The IJACOCO Maven plugin analyzes the tests to select $\mathcal{T}'_{sel}$ at the process-test-classes phase[3], and sets the excludes property of the Surefire plugin[4] to skip the unselected tests. IJACOCO's execution phase happens at the test phase in the Maven lifecycle, where instrumentation is performed by adding the javaagent argument to the Surefire plugin's configuration. IJACOCO's collection phase happens as a shutdown hook at the end of test execution. When report generation is needed, IJACOCO invokes the corresponding JaCoCo functionalities at the verify phase in the Maven lifecycle.

6 Evaluation

In this section, we study the following three research questions to assess IJACOCO's performance and correctness:

RQ1. How much code coverage analysis time speedup can we get from IJACOCO compared with JaCoCo?

---

[3]https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
[4]https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html

Table 1: Repositories used in our evaluation.

Repository	First Ver.	#Ver.	Code Base Size		Test Suite		
			#File	LOC	#Class	#Method	Time [s]
asterisk-java	3576c01f	51	814	57,208	46	252	32.39
commons-beanutils	55a786a3	51	243	31,831	94	1,273	9.84
commons-codec	3c212236	51	133	22,417	58	1,084	44.92
commons-collections	c46666c5	51	536	62,956	168	24,612	25.72
commons-compress	2ed55677	51	356	44,911	137	1,065	24.19
commons-configuration	b84a0ef5	51	467	69,625	169	2,831	30.75
commons-dbcp	8e4a5652	51	109	23,111	32	580	102.36
commons-imaging	c5ca63fe	51	490	38,664	111	573	38.17
commons-io	d3137782	51	250	30,500	106	1,362	46.51
commons-lang	b41e9181	51	333	77,448	148	4,091	27.78
commons-math	aeb21280	51	1,488	191,967	477	6,557	109.81
commons-net	f4bc1441	51	273	28,246	44	301	76.35
commons-pool	1ff0aa0f	51	92	14,395	22	290	330.99
fastjson	30404ab3	51	2,936	177,932	2,278	4,869	83.82
finmath-lib	fbcac3da	51	1,166	101,906	90	496	1,479.98
gerrit-events	5201a56f	51	98	6,649	21	109	26.44
HikariCP	5d1ed1c6	51	90	12,159	37	144	112.40
lmdbjava	7cf8f02f	51	45	4,914	8	105	32.60
LogicNG	60dcb918	51	343	39,532	116	878	452.85
rxjava-extras	11a083ed	51	127	12,898	48	686	94.13
sdk-rest	a800949a	51	626	46,621	24	361	250.40
tabula-java	8247954b	51	52	6,544	15	199	89.14
Σ	-	-	11,067	1,102,434	4,249	52,718	3,521.54

RQ2. How does IJACoCo's test selection rate compare to Ekstazi?
RQ3. How much time does each phase of IJACoCo take?
RQ4. Is IJACoCo correct, i.e., producing the same code coverage results as JaCoCo?

We first describe the subject repositories used in our evaluation (§6.1), then the experimental setup (§6.2), and finally present the results and answer the research questions (§6.3).

6.1 Subjects

We reused the list of repositories and versions in the evaluation of a recent related work on RTS [42], which includes 23 open-source Java repositories and 51 versions per repository. However, we found that email-ext-plugin uses a mocking library that was incompatible with Ekstazi (and thus IJACoCo which is built on top of it) in half of its versions, and thus we excluded this repository. Table 1 lists the remaining 22 subject repositories used in our evaluation, as well as their first (oldest) versions, and metrics of their codebase and test suites. Note that the 51 versions for each repository were selected such that there exists at least a code change at bytecode level (e.g., excluding simple comment changes) between two versions [42]. In total, our evaluation subject set involves 1,122 versions, 1.1M lines of code, and 4,249 test classes. All of the repositories are using the Maven build system.

6.2 Experimental Setup

For each repository, we clone it from GitHub, and then for each version in the selected 51 versions, we checkout to that version, enable one of {IJACoCo, JaCoCo, Ekstazi} or none of them (which we

call RetestAll), and then execute tests using the Maven command `mvn clean test`. We measure the *end-to-end time* of the Maven command, which includes all three phases of IJACoCo and Ekstazi and all steps of JaCoCo. For IJACoCo and JaCoCo, we store the coverage data at each version and report *line-level code coverage* (line coverage) metric in this paper as a representation of the code coverage results. For IJACoCo and Ekstazi, we record the number of tests they selected to compute their *test selection rate* as the number of selected tests divided by the total number of tests.

We enabled IJACoCo, JaCoCo, and Ekstazi by using their Maven plugins. Specifically, we insert a Maven profile into the repository's build configuration file (pom.xml) that adds the corresponding plugin to the Maven build lifecycle. To fairly compare the performance of IJACoCo and JaCoCo, we forked the JaCoCo tool at the same version as the one IJACoCo was built on, and renamed the tool name to bjacoco to avoid interference with existing JaCoCo configurations; then, we disable any existing JaCoCo plugin if the repository has it set up in the build configuration file. For all three tools and RetestAll, we disabled Maven plugins that are not relevant for our experiments, e.g., checkstyle and javadoc.

Due to inevitable test flakiness [13, 34, 35, 44, 52] in complicated codebase, the run time and code coverage metrics may vary across runs (i.e., the execution of tests may take different execution path and result in slightly different code coverage). To mitigate this, we run each experiment 5 times and report the average run time. Moreover, we excluded the flaky tests whose outcomes change or code coverage fluctuate dramatically (e.g., due to their pre-conditions being undeterministically met or not) across the runs. When comparing the code coverage and run time of JaCoCo and IJACoCo,

Table 2: End-to-end time of RetestAll, Ekstazi, JaCoCo, and ıJaCoCo, which are averaged and summed over 51 versions; and speedup of ıJaCoCo compared to JaCoCo. All the time differences between JaCoCo and ıJaCoCo are statistically significant with 95% confidence level.

| Repository | Time [s] | | | | | | | | Speedup |
| | RetestAll | | Ekstazi | | JaCoCo | | ıJaCoCo | | ıJaCoCo |
	avg	Σ	avg	Σ	avg	Σ	avg	Σ	avg
asterisk-java	32.56	1,660.31	11.04	563.02	34.57	1,763.25	12.80	652.89	2.70
commons-beanutils	9.95	507.50	75.65	3,858.11	11.03	562.28	12.37	630.75	0.89
commons-codec	43.95	2,241.52	12.38	631.21	45.17	2,303.74	18.96	967.08	2.38
commons-collections	25.99	1,325.52	12.80	653.05	28.26	1,441.07	20.28	1,034.04	1.39
commons-compress	51.53	2,628.12	20.88	1,065.04	65.93	3,362.41	46.39	2,366.01	1.42
commons-configuration	30.67	1,564.07	21.03	1,072.38	34.38	1,753.52	31.16	1,588.92	1.10
commons-dbcp	107.80	5,498.03	39.52	2,015.29	101.08	5,155.07	63.65	3,246.08	1.59
commons-imaging	38.00	1,937.97	21.72	1,107.63	41.88	2,135.98	29.38	1,498.40	1.43
commons-io	47.42	2,418.42	78.67	4,012.30	49.19	2,508.59	17.23	878.83	2.85
commons-lang	27.15	1,384.52	12.34	629.09	28.72	1,464.68	17.08	871.31	1.68
commons-math	98.81	5,039.51	32.78	1,671.68	131.67	6,715.13	175.71	8,961.26	0.75
commons-net	76.90	3,921.77	17.29	881.79	78.57	4,007.25	26.26	1,339.08	2.99
commons-pool	336.66	17,169.61	96.13	4,902.83	349.86	17,842.92	153.39	7,822.66	2.28
fastjson	82.76	4,220.88	57.59	2,937.18	96.98	4,945.99	87.46	4,460.33	1.11
finmath-lib	1,550.91	79,096.45	261.91	13,357.26	3,992.06	203,595.02	487.13	24,843.60	8.20
gerrit-events	34.59	1,764.25	34.44	1,756.44	35.99	1,835.46	31.71	1,617.05	1.14
HikariCP	127.57	6,506.06	125.66	6,408.81	129.56	6,607.77	100.06	5,103.12	1.29
lmdbjava	61.14	3,117.97	41.63	2,122.99	76.51	3,901.90	59.16	3,017.30	1.29
LogicNG	522.37	26,640.95	344.20	17,554.41	675.95	34,473.64	797.87	40,691.25	0.85
rxjava-extras	93.91	4,789.44	19.22	980.02	102.53	5,228.95	60.83	3,102.37	1.69
sdk-rest	307.53	15,684.03	317.13	16,173.88	405.39	20,675.11	432.54	22,059.67	0.94
tabula-java	86.64	4,418.67	55.48	2,829.27	97.33	4,963.84	101.03	5,152.66	0.96
avg	172.49	8,797.07	77.70	3,962.89	300.57	15,329.25	126.47	6,450.21	1.86

we conducted statistical significance tests using bootstrap tests [6] with a 95% confidence level.
Environment. We run all experiments on servers with 16 Intel Xeon vCPU cores @2.5GHz, 60GB memory, and running Ubuntu 22.04. We use Java version 8.0.392 and Maven version 3.9.6.

6.3 Results

Table 2 shows the end-to-end time of RetestAll and the three tools used in our evaluation, Ekstazi, JaCoCo, and ıJaCoCo. There are two columns for RetestAll or each tool: the avg column shows the average time across the 51 versions of each repository, and the Σ column shows the total time of all versions. The last column of the table computes the speedup of ıJaCoCo compared to JaCoCo, which is JaCoCo's end-to-end time divided by ıJaCoCo's end-to-end time, averaged across 51 versions. The last row computes the average across all repositories.

We can observe that ıJaCoCo achieves an average speedup of 1.86×. The highest speedup of 8.20× was achieved on the `finmath-lib` repository. This demonstrates the effectiveness of incremental code coverage analysis. Notably, we can see that performing code coverage analysis incurs significant overhead, bringing the average end-to-end time from RetestAll's 172.49s to JaCoCo's 300.57s; but ıJaCoCo is able to reduce this time to 126.47s, which is even shorter than RetestAll. This indicates that adopting incremental code coverage analysis can greatly reduce the CI cost for nowadays software

repositories, where computing code coverage on each version is a common practice.

ıJaCoCo is overall faster than JaCoCo on 17 out of 22 repositories. We inspected the remaining 5 repositories and found that ıJaCoCo did not achieve good performance for them due to either (1) RTS selecting a large number of tests (`commons-beanutils` and `sdk-rest`), (2) many tests depend on the same non-test class, so that changing any test requires executing other tests depending on that class, resulting in much more tests being selected by ıJaCoCo than Ekstazi (`commons-math`, `LogicNG`, and `tabula-java`). Nevertheless, the overhead of applying ıJaCoCo for these repositories (and versions) is not large. Future work can investigate how to reduce the overhead, potentially by turning off incremental analysis when certain patterns of code changes are detected.

We also studied the evolution of ıJaCoCo's performance across versions. Due to the page limitations, we highlight the results on a few repositories; the full results can be found in our replication package. Figure 4a and Figure 4b shows the evolution of ıJaCoCo vs. JaCoCo end-to-end time and ıJaCoCo's speedup across 51 versions of `commons-lang`, respectively. We can see that JaCoCo's time is quite stable across versions, while ıJaCoCo's time varies (but mostly shorter than JaCoCo). On versions where the code change is small, ıJaCoCo's time is also shorter; on the first version and one of the later versions where the code change was large, ıJaCoCo needs to execute almost all tests and build dependency graphs, leading to a bit longer time than JaCoCo. Figure 5a and Figure 5b show
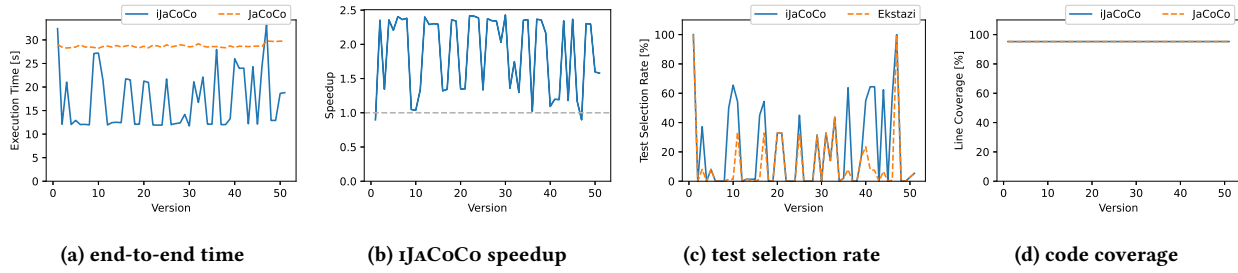
(a) end-to-end time **(b) ɪJᴀCᴏCᴏ speedup** **(c) test selection rate** **(d) code coverage**

Figure 4: Experiment results for commons-lang.

(a) end-to-end time **(b) ɪJᴀCᴏCᴏ speedup** **(c) test selection rate** **(d) code coverage**
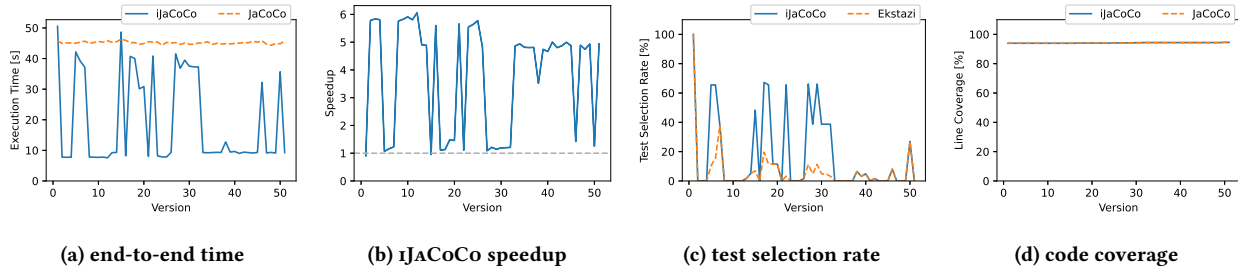
Figure 5: Experiment results for commons-codec.

the same plots for `commons-codec`, and we can see a very similar trend.

In `commons-net` as shown in Figure 6a and Figure 6b, on some of the versions, ɪJᴀCᴏCᴏ does not need to execute any test at all (e.g., due to the class with code change not being covered by any tests), leading to a very high speedup of around 10×. Another interesting case we studied is `fastjson`, shown in Figure 7a and Figure 7b, where ɪJᴀCᴏCᴏ has quite high overhead on several versions (but still achieve an overall speedup of 1.11×). A closer inspection of those versions reveals that when the code change involves adding a test, for example at its 4th version 36e03222[5], ɪJᴀCᴏCᴏ needs to execute almost all the tests as the coverage data of many classes are affected.

> **Answer to RQ1**. ɪJᴀCᴏCᴏ can greatly speed up code coverage analysis when compared to JaCoCo, achieving an average speedup of 1.86× and up to 8.20×.
> Our main observations include: (1) the speedup of ɪJᴀCᴏCᴏ over JaCoCo is correlated with the speedup of Ekstazi over RetestAll; (2) ɪJᴀCᴏCᴏ incurs overhead for collecting dependency graph on the first version (same as RTS) but is usually faster than JaCoCo on subsequent versions; and (3) ɪJᴀCᴏCᴏ is more effective when the code change is small.

The columns 2–3 of Table 3 shows the test selection rate of Ekstazi and ɪJᴀCᴏCᴏ, averaged across all versions. As explained in §4, ɪJᴀCᴏCᴏ needs to select more tests than Ekstazi to support the correct collection of coverage data. On average, ɪJᴀCᴏCᴏ selects 47.34% of the tests, and Ekstazi selects 20.63% of the tests. Figures 4c,

5c, 6c, and 7c show the evolution of test selection rate of ɪJᴀCᴏCᴏ vs. Ekstazi on the four repositories we highlighted earlier. We can see that on some versions, ɪJᴀCᴏCᴏ selects the same number of tests as Ekstazi, while on others it needs to select more tests.

> **Answer to RQ2**. ɪJᴀCᴏCᴏ selects 47.34% tests on average, while Ekstazi selects 20.63%; ɪJᴀCᴏCᴏ needs to select approximately twice as many tests as Ekstazi to support the correct collection of coverage data.

The columns 4–7 of Table 3 shows the time taken by different phases when using ɪJᴀCᴏCᴏ. Specifically, the end-to-end time can be broken down into four phases: (1) the compilation phase of the build system (for compiling the source code and tests), which takes 38.05s or 25.9% on average; (2) the analysis phase of ɪJᴀCᴏCᴏ (§5.1), which takes 1.53s or 1.0% on average; (3) the execution and collection phases of ɪJᴀCᴏCᴏ (§5.2 and §5.3; the two phases are interleaved and thus can only be measured together), which takes 106.34s or 72.5% on average; and (4) the coverage report generation phase (§5.3), which takes 0.84s or 0.6% on average.

> **Answer to RQ3**. The majority of ɪJᴀCᴏCᴏ's end-to-end time is spent on compilation (25.9%) and the execution and collection phases (72.5%). The analysis phase (1.0%) and the report generation (0.6%) introduce a small overhead.
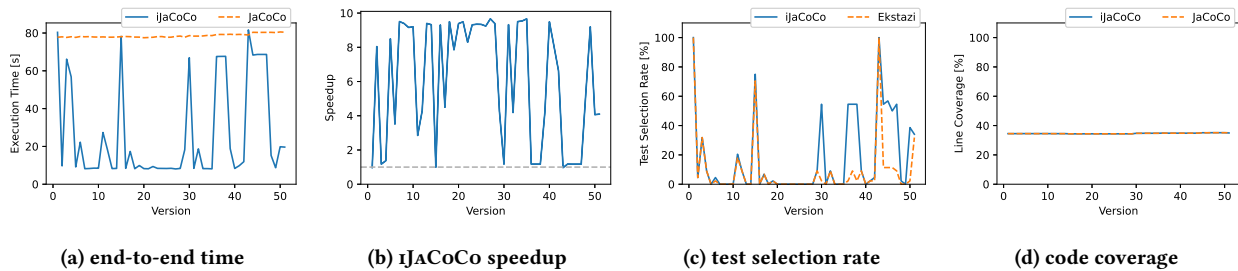
Table 4 compares the line coverage of JaCoCo and ɪJᴀCᴏCᴏ. We report the minimum, maximum, and average line coverage of JaCoCo and ɪJᴀCᴏCᴏ across 51 versions of each repository, as they represent an aggregated distribution of the coverage results. As we can see, most of the numbers are the same or very close. The
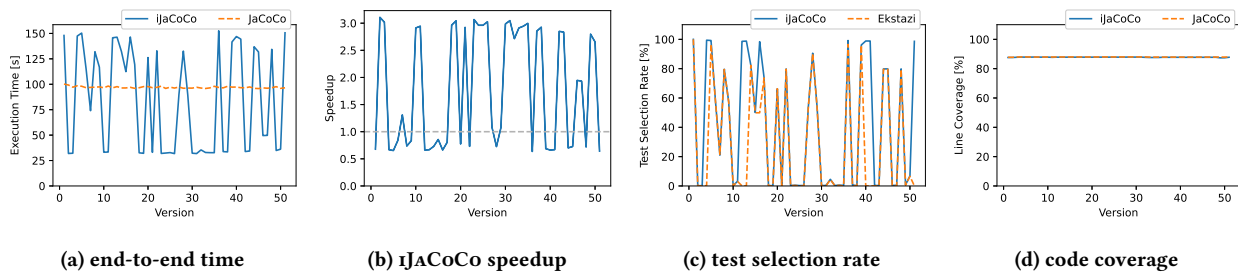
---

[5]Code diff at https://github.com/alibaba/fastjson/compare/12d92f61...36e03222

Table 3: Columns 2–3: test selection rate of Ekstazi and ɪJᴀCᴏCᴏ. Columns 4–7: the time of different phases when using ɪJᴀCᴏCᴏ, and their percentage of ɪJᴀCᴏCᴏ's total end-to-end time.

Repository	Selection Rate [%]		ɪJᴀCᴏCᴏ Phase Time [s]			
	Ekstazi	ɪJACOCO	Compilation	Analysis	Execution + Collection	Report
asterisk-java	5.97	5.97	9.87 (72.8%)	0.42 (3.1%)	2.04 (15.1%)	1.22 (9.0%)
commons-beanutils	12.47	85.09	9.49 (75.4%)	0.40 (3.2%)	2.22 (17.6%)	0.48 (3.8%)
commons-codec	6.34	17.28	7.70 (41.5%)	0.39 (2.1%)	10.00 (53.9%)	0.45 (2.4%)
commons-collections	7.53	34.60	11.09 (56.1%)	0.49 (2.5%)	7.21 (36.5%)	0.98 (5.0%)
commons-compress	20.31	48.73	10.29 (22.4%)	0.58 (1.3%)	34.32 (74.6%)	0.84 (1.8%)
commons-configuration	17.89	48.01	11.73 (36.7%)	1.23 (3.9%)	18.10 (56.7%)	0.89 (2.8%)
commons-dbcp	18.59	31.62	8.21 (13.5%)	0.63 (1.0%)	51.48 (84.6%)	0.50 (0.8%)
commons-imaging	21.97	32.70	9.43 (31.4%)	0.42 (1.4%)	19.22 (64.1%)	0.93 (3.1%)
commons-io	8.01	11.56	8.51 (50.5%)	0.52 (3.1%)	7.27 (43.2%)	0.55 (3.3%)
commons-lang	11.58	22.47	10.47 (63.5%)	0.49 (3.0%)	4.65 (28.2%)	0.87 (5.3%)
commons-math	9.76	96.39	19.83 (10.9%)	0.81 (0.4%)	159.32 (87.6%)	2.02 (1.1%)
commons-net	9.40	18.00	7.64 (29.5%)	0.43 (1.7%)	17.13 (66.2%)	0.68 (2.6%)
commons-pool	13.55	34.49	7.71 (5.1%)	0.39 (0.3%)	141.78 (94.4%)	0.36 (0.2%)
fastjson	28.59	41.41	28.57 (30.8%)	5.79 (6.2%)	57.18 (61.6%)	1.23 (1.3%)
finmath-lib	8.13	7.97	9.91 (2.3%)	0.48 (0.1%)	424.00 (97.3%)	1.59 (0.4%)
gerrit-events	19.51	54.02	6.73 (21.5%)	0.81 (2.6%)	23.40 (74.8%)	0.36 (1.2%)
HikariCP	47.99	59.27	8.46 (8.5%)	0.94 (0.9%)	90.22 (90.2%)	0.44 (0.4%)
lmdbjava	28.10	77.39	6.93 (11.7%)	0.38 (0.6%)	51.56 (87.1%)	0.30 (0.5%)
LogicNG	30.46	84.99	9.03 (1.1%)	0.64 (0.1%)	813.74 (98.7%)	0.92 (0.1%)
rxjava-extras	8.77	47.06	6.67 (11.1%)	0.64 (1.1%)	52.16 (86.8%)	0.60 (1.0%)
sdk-rest	85.13	87.17	10.79 (2.8%)	3.92 (1.0%)	371.25 (95.8%)	1.62 (0.4%)
tabula-java	33.72	95.38	6.63 (7.0%)	0.49 (0.5%)	87.59 (92.2%)	0.34 (0.4%)
avg	20.63	47.34	38.05 (25.9%)	1.53 (1.0%)	106.34 (72.5%)	0.84 (0.6%)

(a) end-to-end time   (b) ɪJᴀCᴏCᴏ speedup   (c) test selection rate   (d) code coverage

Figure 6: Experiment results for commons-net.

(a) end-to-end time   (b) ɪJᴀCᴏCᴏ speedup   (c) test selection rate   (d) code coverage

Figure 7: Experiment results for fastjson.

Table 4: The line coverage of JaCoCo and ɪJaCoCo and the correctness of ɪJaCoCo. Columns 2–7: the minimum, maximum, and average line coverage of JaCoCo and ɪJaCoCo across 51 versions; diff: the average difference between JaCoCo's and ɪJaCoCo's coverage; exact same: whether JaCoCo and ɪJaCoCo is exactly the same across all versions; no stat sign diff: whether the difference between JaCoCo's and ɪJaCoCo's coverage is not statistically significant.

| Repository | Coverage [%] | | | | | | | Correctness | |
| | JaCoCo | | | ɪJaCoCo | | | | exact | no stat |
	min	max	avg	min	max	avg	diff	same	sign diff
asterisk-java	13.5	14.1	14.0	13.5	14.1	14.0	0.01	✓	✓
commons-beanutils	71.3	72.6	71.6	71.3	72.5	71.6	0.01	✓	✓
commons-codec	93.9	94.5	94.2	93.9	94.5	94.2	0.00	✓	✓
commons-collections	87.8	88.7	88.1	87.8	88.7	88.1	0.01	✓	✓
commons-compress	85.9	87.3	87.0	85.9	87.3	87.0	0.01	✓	✓
commons-configuration	88.9	89.5	89.3	88.9	89.5	89.3	0.00	✓	✓
commons-dbcp	40.6	64.1	60.4	40.7	64.1	60.5	0.03	✓	✓
commons-imaging	73.5	74.8	73.9	73.5	74.8	73.9	0.00	✓	✓
commons-io	88.1	88.9	88.5	88.1	88.9	88.5	0.02	✓	✓
commons-lang	95.2	95.3	95.3	95.2	95.3	95.3	0.00	✓	✓
commons-math	90.3	90.9	90.5	90.3	90.9	90.5	0.00	✓	✓
commons-net	34.4	35.1	34.6	34.4	35.1	34.6	0.01	✓	✓
commons-pool	84.8	85.1	85.0	84.8	85.1	85.0	0.03	✓	✓
fastjson	87.8	88.0	87.9	87.6	88.1	87.9	0.05	✓	✓
finmath-lib	35.3	37.4	36.6	35.5	37.5	36.7	0.10	✓	✓
gerrit-events	36.7	51.4	41.7	36.7	51.4	41.7	0.03	✓	✓
HikariCP	78.4	83.3	80.4	78.4	83.4	80.5	0.16		✓
lmdbjava	89.9	91.3	90.6	89.9	91.3	90.6	0.00	✓	✓
LogicNG	94.7	95.1	95.0	94.7	95.1	95.0	0.00	✓	✓
rxjava-extras	67.9	69.3	68.8	67.9	69.3	68.8	0.06	✓	✓
sdk-rest	62.8	66.3	65.2	62.8	66.3	65.2	0.01	✓	✓
tabula-java	82.2	83.7	82.9	82.2	83.7	82.9	0.00	✓	✓

diff column shows the difference between JaCoCo's and ɪJaCoCo's average line coverage. Most of the differences are below 0.10%, and the largest difference of 0.16% happens on the `HikariCP` repository.

These small differences do not necessarily indicate that ɪJaCoCo is incorrect, as code coverage may change if test execution took different paths due to test flakiness and test order dependency. For example, in `fastjson` version `7ffa2a01`, ɪJaCoCo reports that `SerializeWriter.java`'s lines 1977–1998 are covered when running the selected 483 tests, while JaCoCo reports that the same lines of code are not covered when running all 2,278 tests. To determine whether the coverage difference is due to the different set of tests or bugs in ɪJaCoCo, we collected coverage again with JaCoCo but only on the selected tests, and found that those lines of code change from "not covered" to "covered". Thus, the coverage difference is due to a test order dependency, where the selected tests would execute more code if the other tests are not executed. If developers write high-quality tests that do not have such test order dependency, ɪJaCoCo will report the exact same code coverage as JaCoCo.

To determine ɪJaCoCo's correctness, we considered two criteria: (1) exact same, ɪJaCoCo is correct if for all versions, the coverage difference between JaCoCo and ɪJaCoCo is smaller than a threshold of 0.10%; (2) no stat sign diff, ɪJaCoCo is correct if the difference between JaCoCo's and ɪJaCoCo's coverage is not statistically significant. We found that ɪJaCoCo satisfies the first criterion for all but one repository (`HikariCP`), and satisfies the second criterion for all repositories.

In addition, we also generated line plots to illustrate the evolution of JaCoCo's and ɪJaCoCo's coverage across versions, shown in figures 4d, 5d, 6d, and 7d for four of the repositories, and included in the replication package for all repositories. We can see that the lines of JaCoCo and ɪJaCoCo are overlapping, indicating that they produce almost the same code coverage results across all versions.

> **Answer to RQ4**. We performed rigorous examination and confirmed that ɪJaCoCo is correct, i.e., it produces the same code coverage results as JaCoCo.

7 Threats to Validity

External. We have extensively evaluated ɪJaCoCo on a dataset of 1,122 versions from 22 open-source Java repositories. However, this dataset may not be representative of all Java repositories. To mitigate this threat, we used the same repositories and versions that was used in a prior work on RTS [42]; many repositories used in our study have been used in other prior work on RTS [22, 23, 36, 37].

Flaky tests [13, 34, 35, 44, 52], i.e., the tests that may pass or fail without changing the codebase, and tests that may take different execution paths without changing the codebase are very common in practice. Any code coverage analysis tool, including ɪJaCoCo and JaCoCo, may produce different code coverage results across different runs due to test flakiness. To minimize the impact of flaky tests on our evaluation, we have repeated all our experiments 5 times and reported the average results. The standard deviation

of line coverage across the 5 runs is less than 1% for most of the repositories in our evaluation.

Existing RTS techniques might be unsafe in certain scenarios [70]. Our proposed incremental code coverage analysis technique is correct if the underlying RTS technique is safe (§4.5). When the underlying RTS is unsafe, our technique may miss updating the coverage data for some affected classes, leading to incorrect code coverage results. To mitigate this threat, we implement IJaCoCo with Ekstazi as the underlying RTS tool, which has no known safety issues for software executed within the Java virtual machine.

Internal. Our implementation of IJaCoCo may contain bugs that could impact our conclusions. To mitigate this threat, we tested IJaCoCo against JaCoCo as the baseline and confirmed that they produce the same code coverage results modulo the impact of test flakiness. We also performed many sanity checks and manual inspections on our code and scripts.

Construct. An alternative way to realize incremental code coverage analysis is to change the coverage data to be collected per test instead of for the entire test suite, such to avoid selecting more tests than RTS would select. We have explained why this approach may result in much higher analysis overhead in §4.2, and performed preliminary experiments to confirm this overhead.

The sets of instrumentation performed by code coverage analysis to insert probes and by RTS to track dependencies have similar functionalities and could be combined in theory. That is to say, we could further optimize IJaCoCo by only performing one set of instrumentation to track both code coverage and dependencies. We leave this as future work.

8 Related Work

Change impact analysis. Change impact analysis (CIA) [2, 33, 39, 40] is a technique to identify the potential effects of a change in software. CIA can be used for regression testing by selecting tests that cover the changes [45, 50, 54–56, 62, 69]. IJaCoCo's can be seen as a type of CIA but focuses on identifying the affected code coverage by a code change.

Regression test selection. Regression test selection (RTS) [17, 18, 25, 65] is a technique that selects a subset of tests affected by changes in software, thereby reducing the time and cost of regression testing. RTS's dependency analysis can be performed at different granularities, such as coarser-grained, at the target/module-level [14, 19, 46, 60] or finer-grained, at the class/method/statement-level [23, 27, 57, 58, 66]. Many analysis-based RTS tools [42, 66, 68] are proposed for Java projects, among them, Ekstazi [22, 23] and STARTS [36, 37] are popular tools that track class-level dependencies. Ekstazi tracks dependencies dynamically, whereas STARTS tracks dependencies statically. Recently, researchers have also proposed machine learning (ML)-based RTS techniques [7, 16, 43, 51, 67], which uses data-driven models to predict which tests to select instead of analyzing the dependencies.

IJaCoCo is built on Ekstazi as its RTS component. In theory, any RTS tool can be used to support incremental code coverage analysis, but only safe RTS tools can ensure the correctness of the collected code coverage results. Thus, ML-based RTS tools, which are by nature not safe, are not suitable to be used in incremental code coverage analysis.

Applications of RTS. Aside from speeding up regression testing, RTS tools can be used in various contexts. DeFlaker [5] leveraged RTS to identify flaky tests by marking as flaky the tests that fail but are not affected by the changes. Li and Shi [41] proposed IncIDFlakies, a technique that analyzes code changes to detect order-dependent flaky tests from newly-introduced code with the help of RTS. Chen and Zhang [9] sped up mutation testing by only re-collecting the mutation testing results of the affected tests. Genetic improvement [26] can also benefit from RTS by only running the affected tests to evaluate the generated patches. Legunsen et al. [38] leveraged RTS-like technique to speed up runtime verification in evolving software systems. Celik et al. [8] used the idea of RTS to perform regression proof selection in verification projects written in Coq. Our work is the first to apply RTS in the context of speeding up code coverage analysis.

Chittimalli and Harrold [11] explored integrating RTS and code coverage analysis, but for a different purpose than our work: their goal was to speedup RTS by only instrumenting the necessary probes to the code elements covered by the tests. Since reporting code coverage is not their focus, the coverage data is not in the typical format that code coverage analysis techniques use. Specifically, they maintained the set of executed probes per test, which we discussed in §4.2 why this is not desirable due to the overhead of merging coverage data.

Code coverage analysis. Code coverage measures the quality of tests and shows the percentage of code executed by the tests [28, 32]. Code coverage analysis tools [59, 64] capture the coverage and generate reports by instrumenting the code and tracking the execution. However, the overhead of instrumentation and analyzing coverage data to generate reports can be high, especially for large-scale code base [1, 30]. We proposed to speed up code coverage analysis by incrementally computing the coverage data only for the part of codebase affected by the changes, and implemented our idea as IJaCoCo based on a popular industry-level code coverage analysis tool for Java, JaCoCo [61].

9 Conclusion

We proposed the first incremental code coverage analysis technique and its implementation, IJaCoCo, for collecting code coverage in the context of software evolution. The key difference between IJaCoCo and prior work is that it selects a minimal set of tests, saving the costly test execution time, and provides an accurate coverage report upon code changes. We proved the correctness of incremental code coverage analysis and demonstrated the correctness of IJaCoCo. Our evaluation showed that IJaCoCo can significantly speed up code coverage analysis by 1.86× compared to the industrial standard code coverage tool JaCoCo.

Acknowledgments

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 *International Conference on Software Engineering*. 736–745. https://doi.org/10.1145/1985793.1985897

[2] Khubaib Amjad Alam, Rodina Ahmad, Adnan Akhunzada, Mohd Hairul Nizam Md Nasir, and Samee U Khan. 2015. Impact Analysis and Change Propagation in Service-Oriented Enterprises: A Systematic Review. *Information Systems* 54 (2015), 43–73. https://doi.org/10.1016/j.is.2015.06.003

[3] Oracle and/or its affiliates. 2024. *java.lang.instrument.* https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

[4] Carlos Arguelles, Marko Ivanković, and Adam Bender. 2020. *Google Testing Blog: Code Coverage Best Practices.* https://testing.googleblog.com/2020/08/code-coverage-best-practices.html

[5] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *International Conference on Software Engineering*. 433–444. https://doi.org/10.1145/3180155.3180164

[6] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An Empirical Investigation of Statistical Significance in NLP. In *Empirical Methods in Natural Language Processing*. 995–1005.

[7] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3377811.3380369

[8] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. iCoq: Regression Proof Selection for Large-Scale Verification Projects. In *Automated Software Engineering*. 171–182. https://doi.org/10.1109/ase.2017.8115630

[9] Lingchao Chen and Lingming Zhang. 2018. Speeding Up Mutation Testing via Regression Test Selection: An Extensive Study. In *International Conference on Software Testing, Verification, and Validation*. 58–69. https://doi.org/10.1109/icst.2018.00016

[10] John Joseph Chilenski and Steven P Miller. 1994. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal* 9, 5 (1994), 193–200. https://doi.org/10.1049/sej.1994.0025

[11] Pavan Kumar Chittimalli and Mary Jean Harrold. 2009. Recomputing Coverage Information to Assist Regression Testing. *Transactions on Software Engineering* 35, 4 (2009), 452–469. https://doi.org/10.1109/TSE.2009.4

[12] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk.* Pearson Education.

[13] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *International Symposium on the Foundations of Software Engineering*. 830–840. https://doi.org/10.1145/3338906.3338945

[14] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910

[15] William R Elmendorf. 1969. Controlling the Functional Testing of an Operating System. *Transactions on Systems Science and Cybernetics* 5, 4 (1969), 284–290. https://doi.org/10.1109/tssc.1969.300221

[16] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *International Symposium on Software Testing and Analysis*. 491–504. https://doi.org/10.1145/3460319.3464834

[17] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology* 52, 1 (2010), 14–30. https://doi.org/10.1016/j.infsof.2009.07.001

[18] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In *Empirical Software Engineering and Measurement*. 22–31. https://doi.org/10.1145/1414004.1414011

[19] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *International Conference on Software Engineering, Companion*. 11–20. https://doi.org/10.1145/2889160.2889222

[20] Phyllis G Frankl and Stewart N Weiss. 1993. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *Transactions on Software Engineering* 19, 8 (1993), 774–787. https://doi.org/10.1109/32.238581

[21] GitHub. 2024. *Network Dependents, jacoco/jacoco.* https://github.com/jacoco/jacoco/network/dependents

[22] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *International Conference on Software Engineering*, Vol. 2. 713–716. https://doi.org/10.1109/icse.2015.230

[23] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection With Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784

[24] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-based Comparisons of Non-adequate Test Suites. *Transactions on Software Engineering and Methodology* 24, 4 (2015), 1–33. https://doi.org/10.1145/2660767

[25] Todd L Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 2001. An Empirical Study of Regression Test Selection Techniques. *Transactions on Software Engineering and Methodology* 10, 2 (2001), 184–208. https://doi.org/10.1145/367008.367020

[26] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing Genetic Improvement of Software With Regression Test Selection. In *International Conference on Software Engineering*. 1323–1333. https://doi.org/10.1109/icse43902.2021.00120

[27] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. *ACM Sigplan Notices* 36, 11 (2001), 312–326. https://doi.org/10.1145/504311.504305

[28] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *International Conference on Software Quality, Reliability and Security*. 151–156. https://doi.org/10.1109/qrs.2015.30

[29] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Automated Software Engineering*. 426–437. https://doi.org/10.1145/2970276.2970358

[30] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *International Symposium on the Foundations of Software Engineering*. 955–963. https://doi.org/10.1145/3338906.3340459

[31] Yong Woo Kim. 2003. Efficient Use of Code Coverage in Large-Scale Software Development. In *Conference of the Centre for Advanced Studies on Collaborative Research*. 145–155. https://doi.org/10.5555/961322.961347

[32] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code Coverage and Test Suite Effectiveness: Empirical Study With Real Bugs in Large Systems. In *International Conference on Software Analysis, Evolution and Reengineering*. 560–564. https://doi.org/10.1109/saner.2015.7081877

[33] Maria Kretsou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Ignatios Deligiannis, and Vassilis C Gerogiannis. 2021. Change Impact Analysis: A Systematic Mapping Study. *Journal of systems and software* 174 (2021), 110892. https://doi.org/10.1016/j.jss.2020.110892

[34] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *International Conference on Software Engineering*. 1471–1482. https://doi.org/10.1145/3377811.3381749

[35] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *International Conference on Software Testing, Verification, and Validation*. 312–322. https://doi.org/10.1109/icst.2019.00038

[36] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on the Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361

[37] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710

[38] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Rosu, and Darko Marinov. 2019. Techniques for Evolution-Aware Runtime Verification. In *ICST*. 300–311. https://doi.org/10.1109/icst.2019.00037

[39] Steffen Lehnert. 2011. A Taxonomy for Software Change Impact Analysis. In *International Symposium on the Foundations of Software Engineering, Demonstrations*. 41–50. https://doi.org/10.1145/2024445.2024454

[40] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A Survey of Code-Based Change Impact Analysis Techniques. *Software Testing, Verification and Reliability* 23, 8 (2013), 613–646. https://doi.org/10.1002/stvr.1475

[41] Chengpeng Li and August Shi. 2022. Evolution-Aware Detection of Order-Dependent Flaky Tests. In *International Symposium on Software Testing and Analysis*. 114–125. https://doi.org/10.1145/3533767.3534404

[42] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More Precise Regression Test Selection via Reasoning About Semantics-Modifying Changes. In *International Symposium on Software Testing and Analysis*. 664–676. https://doi.org/10.1145/3597926.3598086

[43] Erik Lundsten. 2019. *EALRTS: A Predictive Regression Test Selection Tool.* Master's thesis. KTH Royal Institute of Technology, Sweden. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-264978

[44] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on the Foundations of Software Engineering*. 643–653. https://doi.org/10.1145/2635868.2635920

[45] Evan Martin and Tao Xie. 2007. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In *ICSE SESS*. 5–5. https://doi.org/10.1109/sess.2007.5

[46] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. 233–242. https://doi.org/10.1109/icse-seip.2017.16

[47] John Micco. 2017. The State of Continuous Integration Testing at Google. In *International Conference on Software Testing, Verification, and Validation.*

[48] Joan C Miller and Clifford J Maloney. 1963. Systematic Mistake Analysis of Digital Computer Programs. *Commun. ACM* 6, 2 (1963), 58–63. https://doi.org/10.1145/366246.366248

[49] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the Performance of Maven's Test Isolation: Experience Report. In *International Symposium on Software Testing and Analysis.* 249–259. https://doi.org/10.1145/3395363.3397381

[50] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging Field Data for Impact Analysis and Regression Testing. *ACM SIGSOFT Software Engineering Notes* 28, 5 (2003), 128–137. https://doi.org/10.1145/949952.940089

[51] Rongqi Pan, Mojtaba Bagherzadeh, Taher A Ghaleb, and Lionel Briand. 2022. Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review. *Empirical Software Engineering* 27, 2 (2022), 1–43. https://doi.org/10.1007/s10664-021-10066-6

[52] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–74. https://doi.org/10.1145/3476105

[53] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. 1993. Coverage Measurement Experience During Function Test. In *International Conference on Software Engineering.* 287–301. https://doi.org/10.1109/icse.1993.346035

[54] Omid Pourgalehdari, Kamal Z Zamli, and Nor Ashidi Mat Isa. 2008. A Meta Level Dynamic Approach to Visualize Impact Analysis for Regression Testing. In *2008 International Conference on Computer and Communication Engineering.* 928–931. https://doi.org/10.1109/ICCCE.2008.4580742

[55] Xiaoxia Ren. 2007. *Change Impact Analysis for Java Programs and Applications.* Rutgers The State University of New Jersey, School of Graduate Studies. https://doi.org/doi:10.7282/T3NG4R1W

[56] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Conference on Systems, Programming, Languages, and Applications: Software for Humanity.* 432–448. https://doi.org/10.1145/1035292.1029012

[57] Gregg Rothermel and Mary Jean Harrold. 1994. A Framework for Evaluating Regression Test Selection Techniques. In *International Conference on Software Engineering.* 201–210. https://doi.org/10.1109/ICSE.1994.296779

[58] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[59] Muhammad Shahid and Suhaimi Ibrahim. 2011. An Evaluation of Test Coverage Tools in Software Testing. In *International Conference on Telecommunication Technology and Applications,* Vol. 5.

[60] August Shi, Suresh Thummalapenta, Shuvendu K Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *International Conference on Software Engineering.* 689–699. https://doi.org/10.1109/icse.2017.69

[61] JaCoCo Team. 2024. *Jacoco/Jacoco: Java Code Coverage Library.* https://github.com/jacoco/jacoco

[62] Di Wang, Bixin Li, and Ju Cai. 2008. Regression Testing of Composite Service: An XBFG-based Approach. In *2008 IEEE Congress on Services Part II.* 112–119. https://doi.org/10.1109/services-2.2008.28

[63] Laurie Williams, Gunnar Kudrjavets, and Nachiappan Nagappan. 2009. On the Effectiveness of Unit Test Automation at Microsoft. In *International Symposium on Software Reliability Engineering.* 81–89. https://doi.org/10.1109/ISSRE.2009.32

[64] Qian Yang, J Jenny Li, and David Weiss. 2006. A Survey of Coverage Based Testing Tools. In *International Workshop on Automation of Software Test.* 99–103. https://doi.org/10.1145/1138929.1138949

[65] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[66] Maruf Hasan Zaber. 2021. *Towards Parallelization of Regression Test Selection.* Master's thesis. University of California, Irvine, USA. https://escholarship.org/uc/item/5bx69070

[67] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection. In *ICSE Workshop on Automation of Software Test.* https://doi.org/10.1145/3524481.3527230

[68] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *International Conference on Software Engineering.* 199–209. https://doi.org/10.1145/3180155.3180198

[69] Sai Zhang, Zhongxian Gu, Yu Lin, and Jianjun Zhao. 2008. Celadon: A Change Impact Analysis Tool for Aspect-Oriented Programs. In *International Conference on Software Engineering, Companion.* 913–914. https://doi.org/10.1145/1370175.1370184

[70] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *International Conference on Software Engineering.* 430–441. https://doi.org/10.1109/ICSE.2019.00056

[71] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (1997), 366–427. https://doi.org/10.1145/267580.267590

A   Code and Data

We have included ɪJᴀCᴏCᴏ (in Java), our experiment scripts (in Python), and data (mostly in CSV format) in this package. Please refer to README.md for how to use the code and where to locate the data.

   We will open-source this replication package upon the acceptance of this paper.

B   Detailed Evaluation Results

We include the plots showing detailed evaluation results in this document.

Figure 8 compares the end-to-end time of ɪJᴀCᴏCᴏ and JaCoCo.

   Figure 9 shows the speedup of ɪJᴀCᴏCᴏ (w.r.t. JaCoCo) and compares it with the speedup of Ekstazi (w.r.t. RetestAll). The two speedup values are mostly consistent.

   Figure 10 compares the test selection rate of ɪJᴀCᴏCᴏ and Ekstazi.

   Figure 11 compares the coverage scores of ɪJᴀCᴏCᴏ and JaCoCo, confirming that ɪJᴀCᴏCᴏ produces the correct code coverage results.
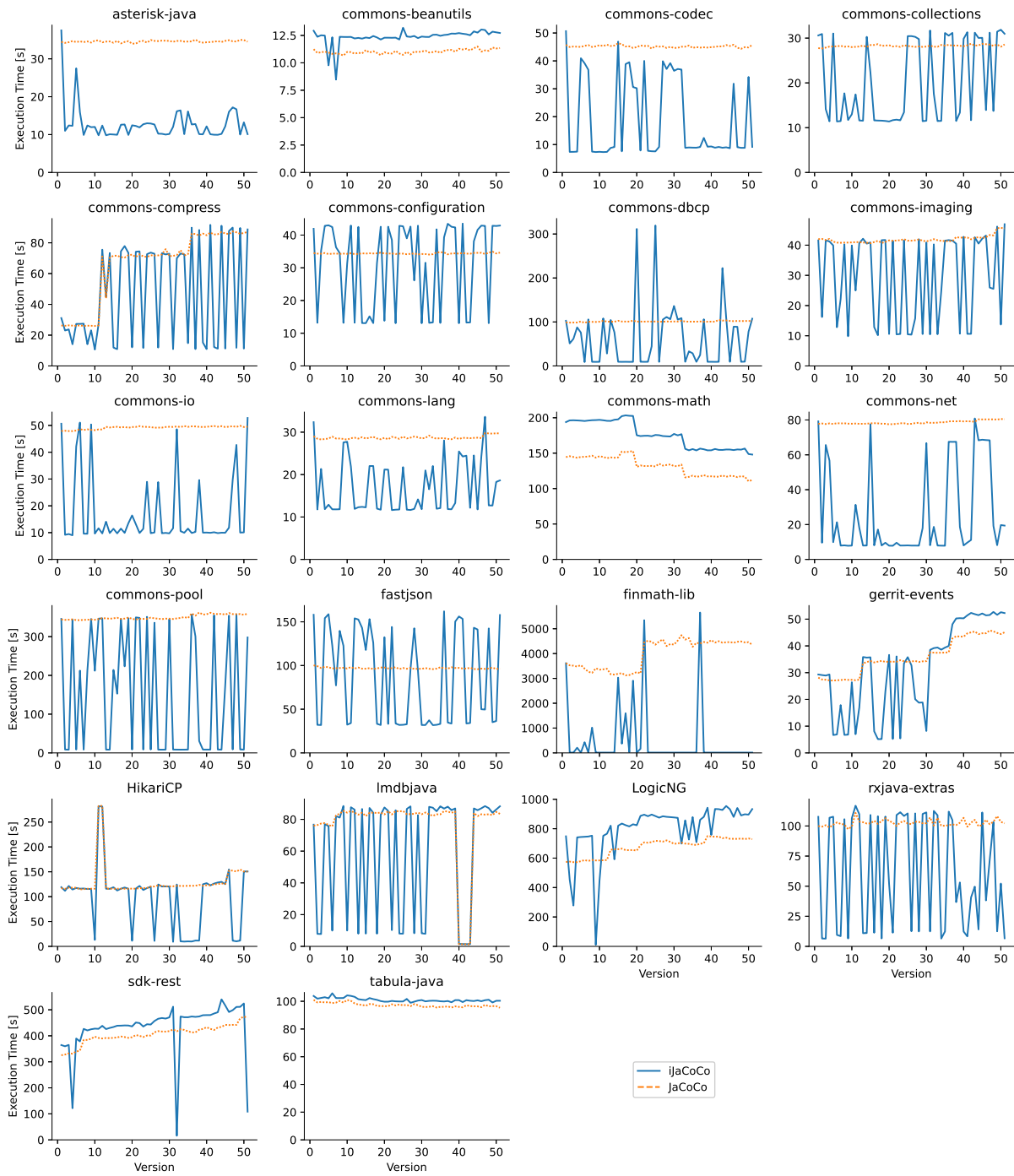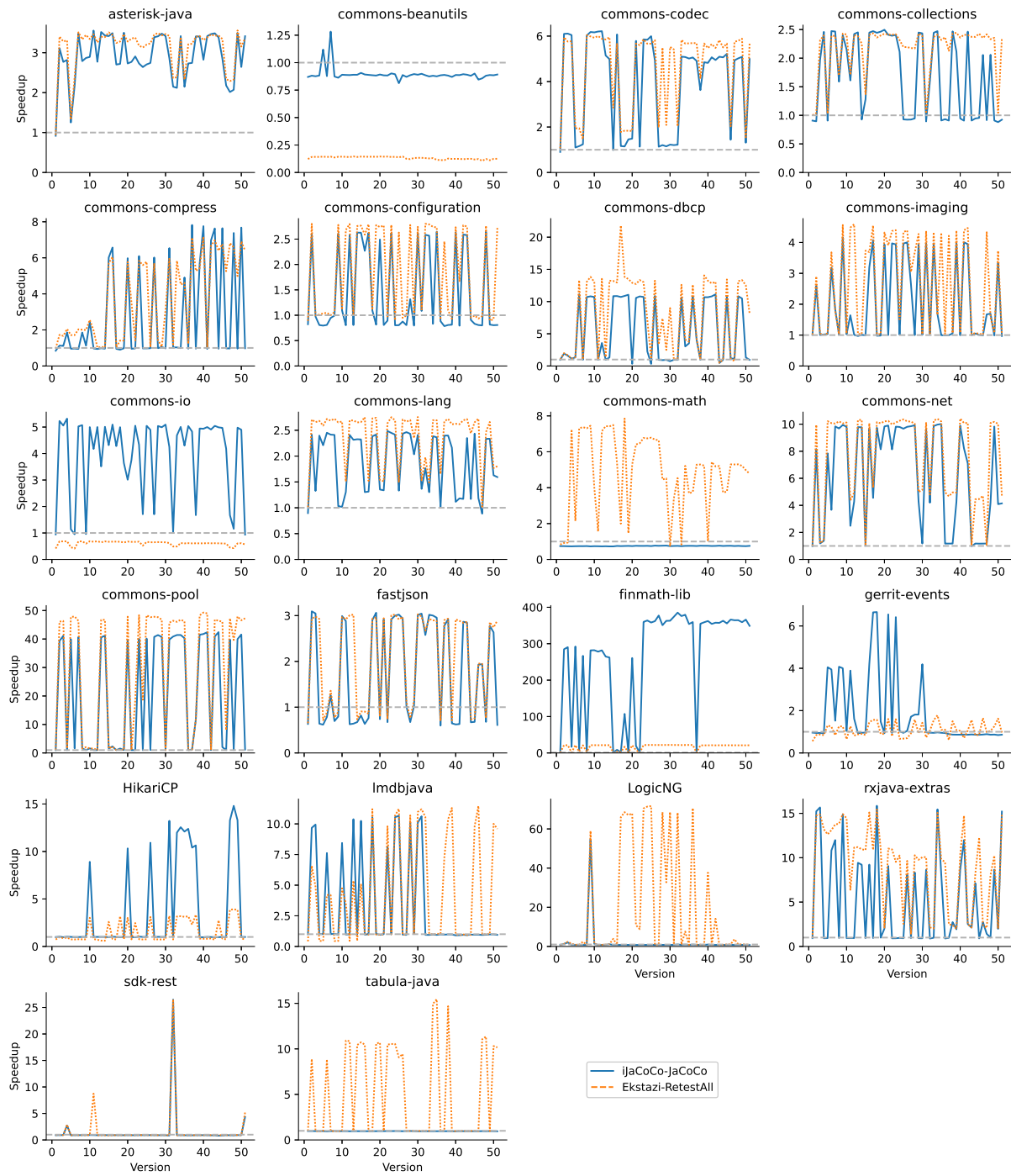
Figure 8: ɪJᴀCᴏCᴏ vs. JᴀCᴏCᴏ end-to-end time.

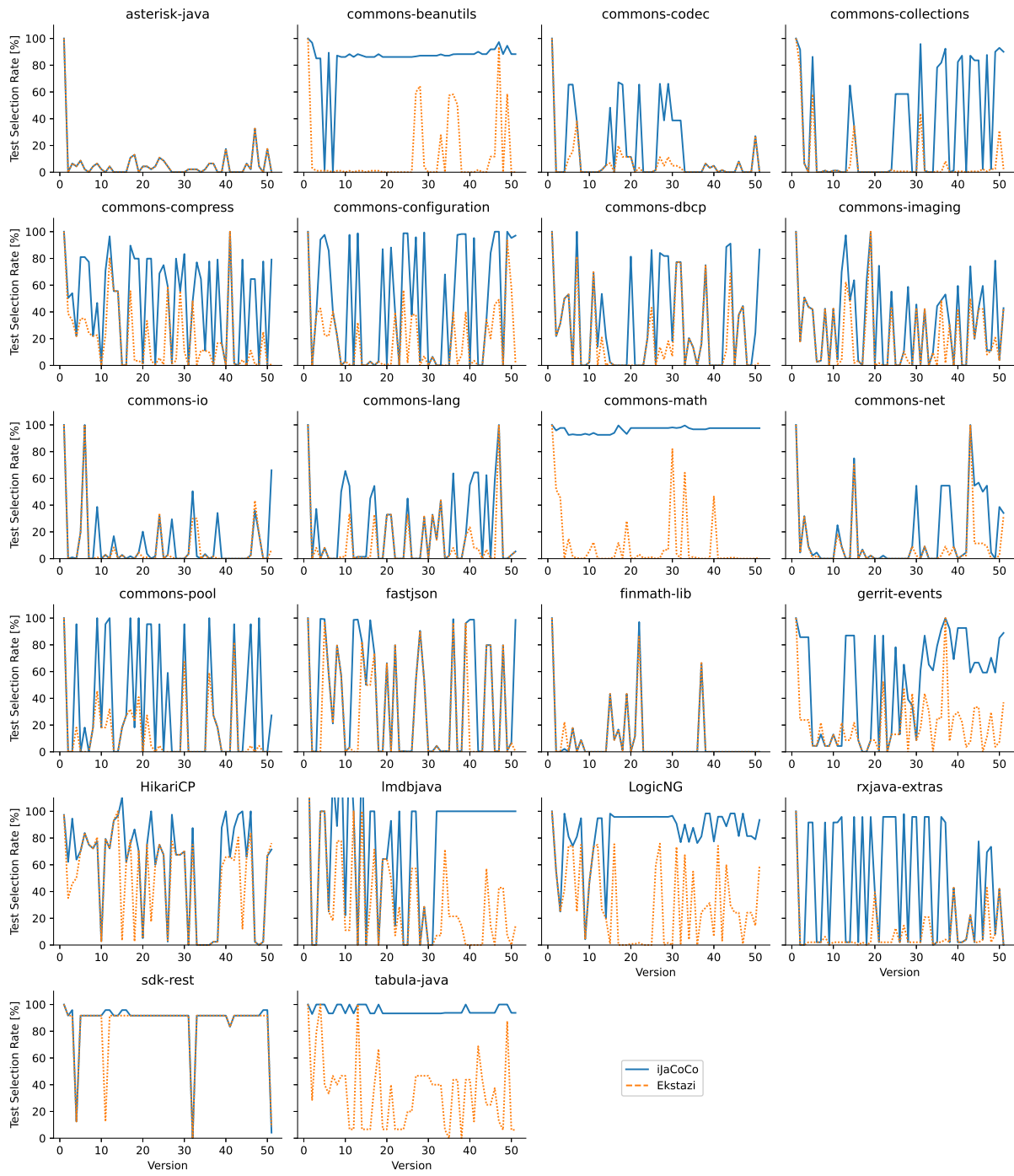**Figure 9: Speedup of ɪJ𝐀C𝐨C𝐨 (w.r.t. JaCoCo) vs. speedup of Ekstazi (w.r.t. RetestAll).**
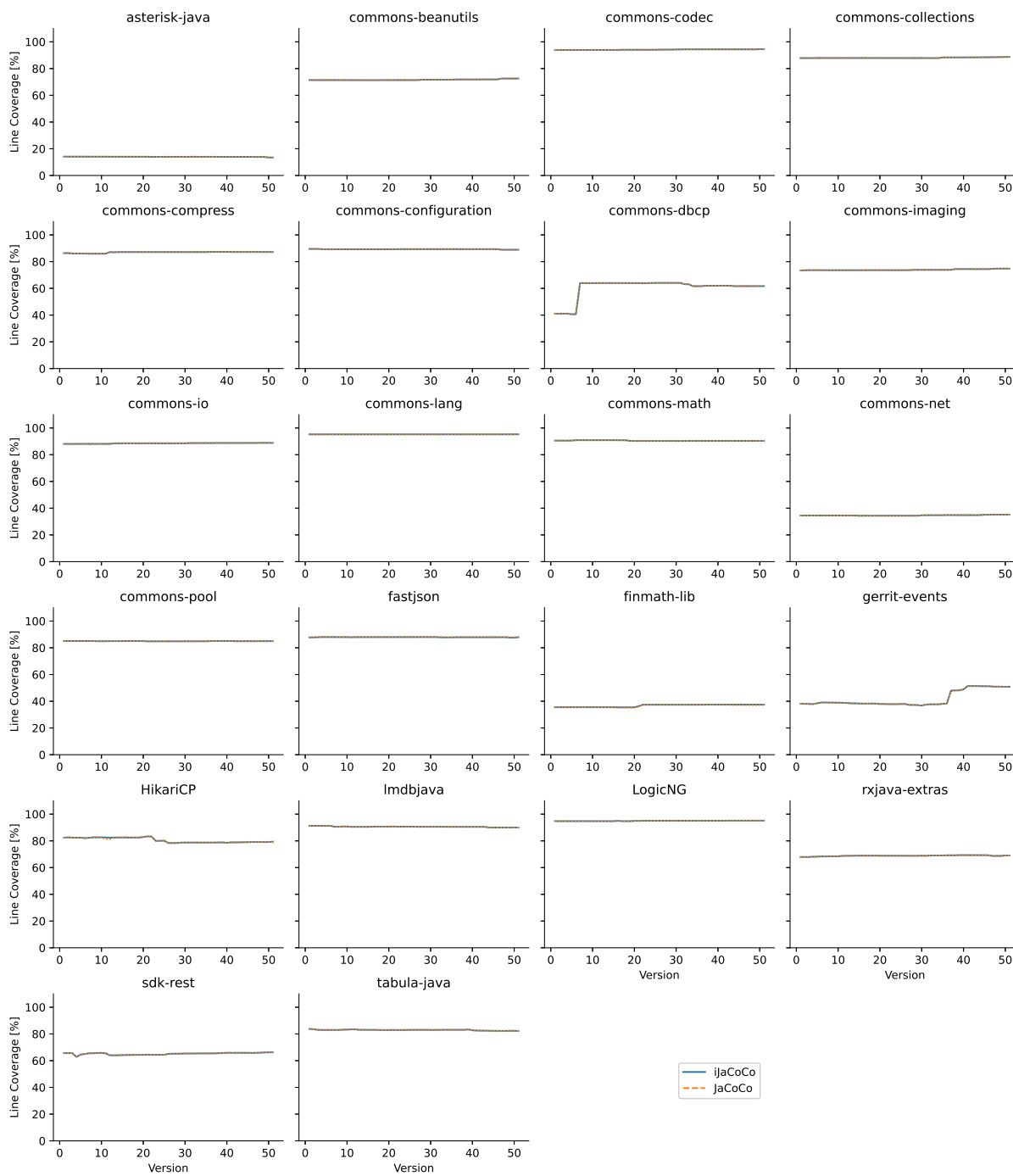
Figure 10: IJaCoCo vs. Ekstazi test selection rate.

Figure 11: ɪJᴀCᴏCᴏ vs. JaCoCo coverage scores.