Practical Regression Test Selection with Dynamic File Dependencies

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov
University of Illinois at Urbana-Champaign
{gliga,eloussi2,marinov}@illinois.edu

ABSTRACT

Regression testing is important but can be time-intensive. One approach to speed it up is regression test selection (RTS), which runs only a subset of tests. RTS was proposed over three decades ago but has not been widely adopted in practice. Meanwhile, testing frameworks, such as JUnit, are widely adopted and well integrated with many popular build systems. Hence, integrating RTS in a testing framework already used by many projects would increase the likelihood that RTS is adopted.

We propose a new, lightweight RTS technique, called Ekstazi, that can integrate well with testing frameworks. Ekstazi tracks dynamic dependencies of tests on files, and unlike most prior RTS techniques, Ekstazi requires no integration with version-control systems. We implemented Ekstazi for Java and JUnit, and evaluated it on 615 revisions of 32 open-source projects (totaling almost 5M LOC) with shorter- and longer-running test suites. The results show that Ekstazi reduced the end-to-end testing time 32% on average, and 54% for longer-running test suites, compared to executing all tests. Ekstazi also has lower end-to-end time than the existing techniques, despite the fact that it selects more tests. Ekstazi has been adopted by several popular open source projects, including Apache Camel, Apache Commons Math, and Apache CXF.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Experimentation

Keywords: Regression test selection, file dependencies

1. INTRODUCTION

Regression testing is important for checking that software changes do not break previously working functionality. However, regression testing is costly as it runs a large number of tests. Some studies [13, 15, 21, 38, 43] estimate that regression testing can take up to 80% of the testing budget and up to 50% of the software maintenance cost. The cost of regression testing increases as software grows. For example, Google observed that their regression-testing system, TAP [20, 55, 57], has had a linear increase in both the number of software changes and the average test-suite execution time, leading to a quadratic increase in the total test-suite-execution time. As a result, the increase is challenging to keep up with even for a company with an abundance of computing resources.

Regression test selection (RTS) is a promising approach to speed up regression testing. Researchers have proposed many RTS techniques (e.g., [23, 27, 31–33, 50, 62]); Engström et al. [22] present a survey of RTS, and Yoo and Harman [59] present a survey of regression testing including RTS. A traditional RTS technique takes four inputs—two software revisions (new and old), test suite at the new revision, and dependency information from the test runs on the old revision—and produces, as output, a subset of the test suite for the new revision. The subset includes the tests that can be affected by the changes; viewed dually, it excludes the tests that cannot be affected by the changes and thus, need not be rerun on the new revision. RTS is *safe* if it guarantees that the subset of selected tests includes *all* tests whose behavior may be affected by the changes.

While RTS was proposed over three decades ago [22, 59], it has not been widely adopted in practice, except for the substantial success of the Google TAP system [20, 55, 57]. Unfortunately, TAP performs RTS only *across projects* (e.g., YouTube depends on the Guava project, so all YouTube tests are run if anything in Guava changes) and provides no benefit *within a project*. However, most developers work on one isolated project at a time rather than on a project from a huge codebase as done at Google. Moreover, our recent empirical study [26] shows that developers who work on such isolated projects frequently perform *manual* RTS, i.e., select to run only some of the tests from their test suite, even when their test suites are relatively fast. In sum, a large number of developers would benefit from an automated RTS technique that could work in practice.

Relevant Problem: We pose the following question: how should researchers design an RTS technique and tool to increase the likelihood that RTS actually be adopted in practice? We note that testing frameworks, such as JUnit, are widely adopted and well integrated with many popular build systems, such as Ant or Maven. For example, our analysis of 666 most active, Maven-based[1] Java projects from GitHub showed that at least 520 (78%) use JUnit (and 59 more

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771784

[1] We cloned 2000 most active Java projects but filtered those that did not use Maven to automate our analysis.

use TestNG, another testing framework). In addition, at least 101 projects (15%) use a code coverage tool, and 2 projects even use a mutation testing tool (PIT [47]). Yet, *no project* used automated RTS. We believe that integrating a lightweight RTS technique with an existing testing framework would likely increase RTS adoption. Ideally, a project that already uses the testing framework could adopt RTS with just a minimal change to its build script, such as `build.xml` or `pom.xml`.

The key requirement for an RTS technique to be adopted is that the *end-to-end* time, on average, is shorter than the time to execute all tests in the testing framework [37, 40]. A typical RTS technique has three phases: the *analysis ($\mathcal{A}$) phase* selects tests to run, the *execution ($\mathcal{E}$) phase* runs the selected tests, and the *collection ($\mathcal{C}$) phase* collects information from the current revision to enable the analysis for the next revision. Most research has evaluated RTS techniques based on the number of selected tests, i.e., implicitly based on the time only for the $\mathcal{E}$ phase; a few papers that do report time (e.g., [45, 58]) measure only $\mathcal{A}$ and $\mathcal{E}$ phases, ignoring the $\mathcal{C}$ phase. To properly compare speedup (or slowdown) of RTS techniques, we believe it is important to consider the end-to-end time that the developer observes, from initiating the test-suite execution for a new code revision until all test outcomes become available.

Lightweight Technique: We propose Ekstazi, a novel RTS technique based on *file dependencies*. Ekstazi is motivated by recent advances in build systems [2, 3, 7–9, 17, 24, 42] and prior work on RTS based on class dependencies [21–23, 34, 36, 45, 52] and external resources [29, 30, 43, 58], as discussed further in Section 7. Unlike most prior RTS techniques based on finer-grained dependencies (e.g., methods or basic blocks), Ekstazi does *not* require integration with version-control systems (VCS): Ekstazi does not explicitly compare the old and new revisions. Instead, Ekstazi computes for each *test entity* (a test method or a test class) what files it depends on; the files can be either executable code (e.g., `.class` files in Java) or external resources (e.g., configuration files). A test need not be run in the new revision if none of its dependent files changed. While we provide no formal proof that Ekstazi is safe, its safety follows directly from the proven safety for RTS based on class dependencies [52] and partial builds based on file dependencies [17].

Robust Implementation: We implement the Ekstazi technique in a tool [25] integrated with JUnit. Our tool handles many features of Java, such as packing of `.class` files in `.jar` archives, comparison of `.class` files using smart checksums (e.g., ignoring debug information), instrumentation to collect dependencies using class loaders or Java agents, reflection, etc. Our tool can work out-of-the-box on any project that uses JUnit. The tool is available at `www.ekstazi.org`.

Extensive Evaluation: We evaluate Ekstazi on 615 revisions of 32 Java projects, ranging from 7,389 to 920,208 LOC and from 83 to 641,534 test methods that take from 8 to 2,565 seconds to execute in the base case, called *RetestAll* (that runs all the tests) [37]. To the best of our knowledge, this is the largest evaluation of any RTS study, and the first to report the end-to-end RTS time, including the $\mathcal{C}$ phase. The experiments show that Ekstazi reduces the end-to-end time 32% on average, 54% for longer-running test suites, compared to RetestAll. Further, Ekstazi reduces the time 47% on average, 66% for longer-running test suites, when the $\mathcal{C}$ phase is performed in a separate, off-line run [15, 18].

```
class TestM {
  void t1() {
    assert new C().m() == 1; }
  void t2() {
    assert new D().m() == 1; }
}

class TestP {
  void t3() {
    assert new C().p() == 0; }
  void t4() {
    assert new D().p() == 4; }
}
```

```
class C {
  C() {}
  int m() {
    /* no method calls */ }
  int p() {
    /* no method calls */ }
}

class D extends C {
  D() {}
  @Override
  int p() {
    /* no method calls */ }
}
```

Figure 1: Example test code and code under test

```
t1 → C.C, C.m
t2 → D.D, C.C, C.m
t3 → C.C, C.p
t4 → D.D, C.C, D.p
```

(a) meth-meth

```
t1 → TestM, C
t2 → TestM, D, C
t3 → TestP, C
t4 → TestP, D, C
```

(b) meth-class

```
TestM → TestM, C, D
TestP → TestP, C, D
```

(c) class-class

Figure 2: Dependencies collected for code in Figure 1

We also compare Ekstazi with FaultTracer [60], a state-of-the-research RTS tool based on fine-grained dependencies, on a few projects that FaultTracer can work on. Not only is Ekstazi faster than FaultTracer, but FaultTracer is, on average, even slower than RetestAll. We discuss why the main result—that Ekstazi is better than FaultTracer in terms of the end-to-end time—is not simply due to FaultTracer being a research prototype but a likely general result. Ekstazi tracks dependencies at our proposed file granularity, whereas FaultTracer uses a finer granularity. While Ekstazi does select more tests than FaultTracer and has a slightly slower $\mathcal{E}$ phase, Ekstazi has much faster $\mathcal{A}$ and $\mathcal{C}$ phases and thus, a lower end-to-end time.

Ekstazi has already been integrated in the main repositories of several open-source projects where it is used on a daily basis, including in Apache Camel [10], Apache Commons Math [11], and Apache CXF [12].

2. EXAMPLE

We use a synthetic example to introduce key terms and illustrate several RTS techniques and their trade-offs. Figure 1 shows code that represents an old project revision: two test classes—`TestM` and `TestP`—contain four test methods—`t1`, `t2`, `t3`, and `t4`—for two classes under test—`C` and `D`.

Executing the tests on this revision can collect the dependencies that relate each *test entity* to a set of *dependent elements*. These elements can be of various granularity; for our example, we use methods and classes. We refer to the granularity of test entities as *selection granularity*—this is the level at which tests are tracked and selected (as test methods or test classes), and we refer to the granularity of dependent elements as *dependency granularity*—this is the level at which changes are determined.

A traditional RTS technique, e.g., FaultTracer [60], using methods for both the selection granularity and the dependency granularity would collect the dependencies as in Figure 2a. Ekstazi uses classes (more generally, files) for the dependency granularity and either methods or classes for the selection granularity. Using methods or classes collects the dependencies as in Figure 2b or Figure 2c, respectively.

In Ekstazi, whenever a test entity depends on `D`, it also depends on `C` (in general, on all superclasses of `D`). Each test entity also depends on its test class, e.g., `t1` depends on

TestM. Finally, this simple example does not show the test code or the code under test accessing any files, but EKSTAZI also tracks files (including directories).

Assume that a new code revision changes only the body of the method D.p and thus only the class D. FaultTracer would select to run only one test, t4. In contrast, EKSTAZI at the method granularity would select two tests, t2 and t4, because they both depend on the changed class D. Moreover, EKSTAZI at the class granularity would select both test classes, TestM and TestP, and thus all four test methods, because both test classes depend on the changed class D.

At a glance, it seems that EKSTAZI cannot be better than the traditional techniques, because EKSTAZI never selects fewer tests. However, our goal is to optimize the end-to-end time for RTS. Although EKSTAZI selects some more tests and thus has a longer execution phase, its use of much coarser dependencies shortens both the analysis and collection. As a result, EKSTAZI has a much lower end-to-end time.

Safely using methods as the dependency granularity is expensive. An RTS technique that just intersects methods that are in the set of dependencies with the changes, as we discussed in our simplified description, is *unsafe*, i.e., it could fail to select some test that is affected by the changes. For example, the new revision could add a method m in class D (that overrides C.m); a naive intersection would not select any test, but the outcome of t2 could change: the execution of this test on the old revision does depend on the existence (or absence) of D.m, although the test could not execute that (non-existent) method [31, 48]. For another example, the new revision could change the value of a final field accessed from the existing method C.m; it would be necessary to collect accessed fields to safely reason about the change and determine which tests should be selected [48, 60].

As a consequence, an RTS technique that uses methods as the dependency granularity could be safer by collecting more dependencies than just used methods (hence making the collection expensive and later selecting more tests, making the execution more expensive), and, more critically, it also needs sophisticated, expensive comparison of the old and new revisions to reason about the changes (hence making the analysis phase expensive). In contrast, an RTS technique that uses classes as the dependency granularity can be safer by simply collecting all used classes (hence speeding up the collection), and more critically, it can use a rather fast check of the new revision that does not even require the old revision (hence speeding up the analysis phase). However, when tests depend not only on the code under test but also on external files [17, 43], collecting only the classes is not safe, and hence EKSTAZI uses files as dependencies.

3. TECHNIQUE AND IMPLEMENTATION

A typical RTS technique has three phases: the *analysis ($\mathcal{A}$) phase* selects what tests to run in the current revision, the *execution ($\mathcal{E}$) phase* runs the selected tests, and the *collection ($\mathcal{C}$) phase* collects information for the next revision. EKSTAZI collects dependencies at the level of files. For each test entity, EKSTAZI saves (in the corresponding *dependency file*[2]) the names and checksums of the files that the entity uses during execution.

Ekstazi technique is safe for *any* code change and *any* change on the file system. The safety of EKSTAZI intuitively follows from the proven safety of RTS based on class dependencies [52] and partial builds based on file dependencies [17]. We leave it as a future work to formally prove that EKSTAZI is safe.

In the rest of the section, we first describe the RTS phases in more detail. We then describe the format in which EKSTAZI saves the dependencies, and describe an optimization that is important to make EKSTAZI practical. We finally describe EKSTAZI integration with a testing framework.

3.1 Analysis ($\mathcal{A}$) Phase

The analysis phase in EKSTAZI is quite simple and thus fast. For each test entity, EKSTAZI checks if the checksums of all used files are still the same. If so, the test entity is not selected. Note that an executable file can remain the same even when its source file changes (e.g., renaming a local variable or using syntactic sugar); dually, the executable file can change even when its source file remains the same (e.g., due to a change in compilation). This checking requires no sophisticated comparisons of the old and new revisions, which prior RTS research techniques usually perform on the source, and in fact it does not even need to analyze the old revision (much like a build system can incrementally compile code just by knowing which source files changed). The only check is if the files remained the same.

EKSTAZI naturally handles newly added test entities: if there is no dependency information for some entity, it is selected. Initially, on the very first run, there is no dependency information for any entity, so they are all selected.

3.2 Execution ($\mathcal{E}$) Phase

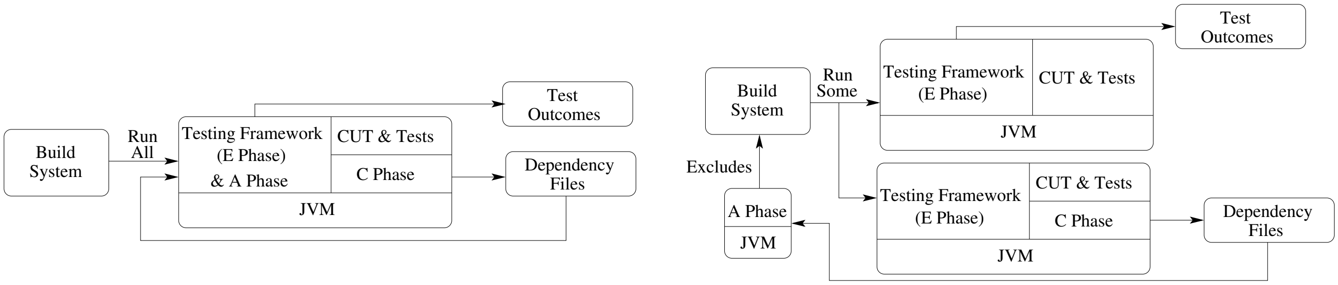
Although one can initiate test execution directly from a testing framework, large projects typically initiate test execution from a build system. Popular build systems (e.g., Ant or Maven) allow the user to specify an includes list of all test classes to execute.

Figure 3 shows two approaches to integrate RTS in a typical Java project. EKSTAZI can work with testing frameworks in both approaches. When tightly integrating the $\mathcal{A}$ and $\mathcal{E}$ phases (Figure 3a), the build system finds all test classes and invokes a testing framework as if all test entities will run; EKSTAZI then checks for each entity if it should be actually run or not. Tightly integrating these two phases simplifies adding EKSTAZI to an existing build: to use EKSTAZI requires only a single change, to replace testingframework.jar (e.g., junit.jar) with ekstazi.jar.

Loosely integrating the $\mathcal{A}$ and $\mathcal{E}$ phases (Figure 3b) can improve performance in some cases. It first determines what test entities *not* to run. This avoids the unnecessary overhead (e.g., loading classes or spawning a new JVM when the build spawns a JVM for each test entity) of preparing to run an entity and finding it should not run. The $\mathcal{A}$ phase makes an excludes list of test *classes* that should not run, and the build system ignores them before executing the tests. EKSTAZI makes an excludes list from previously collected dependency files and excludes test *classes* rather than test *methods* because most build systems support an excludes list of classes. In case of the method selection granularity, the test methods that are not affected are excluded at the beginning of the $\mathcal{E}$ phase.

Figure 3 also shows two approaches to integrate the $\mathcal{E}$ and $\mathcal{C}$ phases. First, the dependencies for the test entities

[2] Note that a "dependency file", which stores dependencies, should not be confused with "dependent files", which are the dependencies themselves.



(a) Tight integration (b) Loose integration

Figure 3: Integration of an RTS technique in a typical build with a testing framework

that were not selected cannot change: these entities are not run and their corresponding dependency files do not change. But the test entities that were selected need to be run to determine if they still pass or fail, and thus to inform the user who initiated the test-suite execution. Because the dependencies for these entities change, the simplest way to update their dependency files is with *one pass* ($\mathcal{AEC}$) that both determines the test outcome and updates the dependency files. However, collecting dependencies has an overhead. Therefore, some settings [15, 18] may prefer to use *two passes*: one without collecting dependencies ($\mathcal{AE}$), just to determine the test outcome and inform the user, and another to also collect the dependencies ($\mathcal{AEC}$). The second pass can be started in parallel with the first or can be performed sequentially later.

3.3 Collection ($\mathcal{C}$) Phase

The collection phase creates the dependency files for the executed test entities. Ekstazi monitors the execution of the tests and the code under test to collect the set of files accessed during execution of each entity, computes the checksum for these files, and saves them in the corresponding dependency file. Ekstazi currently collects *all* files that are either read or written, but it could be even more precise by distinguishing writes that do not create a dependency [28]. Moreover, Ekstazi tracks even files that were attempted to be accessed but did not exist; if those files are added later, the behavior can change.

In principle, we could collect file dependencies by adapting a tool such as Fabricate [24] or Memoize [42]: these tools can monitor any OS process to collect its file dependencies, and thus they could be used to monitor a JVM that runs tests. However, these tools would be rather *imprecise* for two reasons. First, they would not collect dependencies per entity when multiple entities run in one JVM. Second, they would not collect dependencies at the level of `.class` files archived in `.jar` files. Moreover, these tools are not portable from one OS to another, and cannot be easily integrated in a testing framework such as JUnit or a build system such as Maven or Ant.

We implemented the $\mathcal{C}$ phase in Ekstazi as a pure Java library that is called from a testing framework and addresses both points of imprecision. To collect dependencies per test entity, Ekstazi needs to be informed when an entity starts and ends. Ekstazi offers API methods `entityStarted(String name)`, which clears all previously collected dependencies, and `entityEnded(String name)`, which saves all the collected dependencies to an appropriately named dependency file.

When using method selection granularity, due to common designs of testing frameworks, additional steps are needed to properly collect dependencies. Namely, many testing frameworks invoke a constructor of a test class only once, and then invoke `setUp` method(s) before each test method is invoked. Therefore, Ekstazi appends dependencies collected during constructor invocation and `setUp` method(s) to the dependencies collected during the execution of each test method.

To precisely collect accessed files, Ekstazi dynamically instruments the bytecode and monitors the execution to collect both explicitly accessed files (through the `java.io` package) and implicitly accessed files (i.e., the `.class` files that contain the executed bytecode). Ekstazi collects explicitly used files by monitoring all standard Java library methods that may open a file (e.g., `FileInputStream`). Files that contain bytecode for Java classes are not explicitly accessed during execution; instead, a class loader accesses a classfile when a class is used for the first time. Our instrumentation collects a set of objects of the type `java.lang.Class` that a test depends on; Ekstazi then finds for each class where it was loaded from. If a class is not loaded from disk but dynamically created during execution, it need not be tracked as a dependency, because it cannot change unless the code that generates it changes.

Instrumented Code Points: More precisely, Ekstazi instruments the following code points: (1) start of a constructor, (2) start of a static initializer, (3) start of a static method, (4) access to a static field, (5) use of a class literal, (6) reflection invocations, and (7) invocation through `invokeinterface` (bytecode instruction). Ekstazi needs no special instrumentation for the test class: it gets captured as a dependency when its constructor is invoked. Ekstazi also does not instrument the start of instance methods: if a method of class `C` is invoked, then an object of class `C` is already constructed, which captured the dependency on `C`.

3.4 Dependency Format

Ekstazi saves dependencies in a simple format similar to the dependency format of build tools such as Fabricate [24]. For each test entity, Ekstazi saves all the information in a separate dependency file whose name corresponds to the entity name. For example in Figure 2b, Ekstazi creates four files `TestM.t1`, `TestM.t2`, `TestP.t3`, and `TestP.t4`. Saving dependencies from all test entities in one file would save space and could save time for smaller projects, but it would increase time for large projects that often run several test entities in parallel (e.g., spawn multiple JVMs) so using one dependency file would require costly synchronization. In the future, we plan to explore other ways to persist the dependencies, e.g., databases.

3.5 Smart Checksums

EKSTAZI's use of file checksums offers several advantages, most notably (1) the old revision need not be available for the $\mathcal{A}$ phase, and (2) hashing to compute checksums is fast. On top of collecting the executable files (`.class`) from the archives (`.jar`), EKSTAZI can compute the *smart checksum* for the `.class` files. Computing the checksum from the bytecodes already ignores some changes in the source code (e.g., `i++` and `i+=1` could be compiled the same way). The baseline approach computes the checksum from the entire file content, including all the bytecodes.

However, two somewhat different executable files may still have the same semantics in most contexts. For example, adding an empty line in a `.java` file would change the debug info in the corresponding `.class` file, but almost all test executions would still be the same (unless they observe the debug info, e.g., through exceptions that check line numbers). EKSTAZI can ignore certain file parts, such as compile-time annotations and other debug info, when computing the checksum. The trade-off is that the smart checksum makes the $\mathcal{A}$ and $\mathcal{C}$ phases slower (rather than quickly applying a checksum on the entire file, EKSTAZI needs to parse parts of the file and run the checksum on a part of the file), but it makes the $\mathcal{E}$ phase faster (as EKSTAZI selects fewer tests because some dependent files match even after they change).

3.6 Integrating Ekstazi with JUnit

We implemented the EKSTAZI technique in a robust tool for Java and JUnit [25]. We integrated EKSTAZI with JUnit because it is a widely used framework for executing unit tests in Java. Regarding the implementation, our `ekstazi.jar` has to change (dynamically) a part of the JUnit core itself to allow EKSTAZI to skip a test method that should not be run. While JUnit provides listeners that can monitor start and end of tests, currently the listeners cannot change the control-flow of tests. EKSTAZI supports both JUnit 3 and JUnit 4, each with some limitation. For JUnit 3, EKSTAZI supports only methods (not classes) as selection granularity. For JUnit 4, if a project uses a custom runner, EKSTAZI supports only classes (not methods); otherwise, if no custom runner is used, EKSTAZI supports both classes and methods. It is important to note that when EKSTAZI does not support some case, it simply offers no test selection and runs all the tests, as RetestAll.

4. EVALUATION

This section describes an experimental evaluation of EKSTAZI. We (1) describe the projects used in the evaluation, (2) describe the experimental setup, (3) report the RTS results in terms of the number of selected test entities and time, (4) measure benefits of the smart checksum, (5) evaluate the importance of selection granularity, (6) evaluate the importance of dependency granularity by comparing EKSTAZI with FaultTracer [60], and (7) describe a case study of EKSTAZI integration with a popular open-source project. More information about the experiments (e.g., links to the projects used in evaluation) is available at `www.ekstazi.org/research.html`.

We ran all the experiments on a 4-core 1.6 GHz Intel i7 CPU with 4GB of RAM, running Ubuntu Linux 12.04 LTS. We used three versions of Oracle Java 64-Bit Server: 1.6.0_45, 1.7.0_45, and 1.8.0_05. Different versions were necessary as several projects require specific older or newer Java version. For each project, we used the latest revision that successfully compiled and executed all tests.

4.1 Projects

Figure 4 lists the projects used in the evaluation; all 32 projects are open source. The set of projects was created by three undergraduate students who were not familiar with our study. We suggested starting places with larger open-source projects: Apache [1], GitHub [4], and GoogleCode [5]. We also asked that each project satisfies several requirements: (1) has the latest available revision (obtained at the time of the first download) build without errors (using one of three Java versions mentioned above), (2) has at least 100 JUnit tests, (3) uses Ant or Maven to build code and execute tests, and (4) uses SVN or Git version-control systems. The first two requirements were necessary to consider compilable, non-trivial projects, but the last two requirements were set to simplify our automation of the experiments.

Note that EKSTAZI itself does *not* require any integration with VCS, but our experiments do require to automate checking out of various project revisions. To support both Ant and Maven across many project revisions, we do not modify the `.xml` configuration files but replace the appropriate `junit.jar` (in the `lib` for Ant-based projects or in the Maven `.m2` download repo) with our `ekstazi.jar`. From about 100 projects initially considered from the three source-code repositories, two-thirds were excluded because they did not build (e.g., due to syntax errors or missing dependencies), used a different build systems (e.g., Gradle), or had too few tests. The students confirmed that they were able to execute JUnit tests in all selected projects.

Figure 4 tabulates for each project its name, the latest revision available at the time of our first download, the number of revisions that could build out of 20 revisions *before* the specified revision, the total number of lines of code (as reported by sloccount [53]), the number of JUnit test methods and classes averaged across all buildable revisions, and the average (*avg*) and total ($\sum$) time to execute the entire test suite across all buildable revisions. The remaining columns are discussed in the following sections.

The row labeled $\sum$ at the bottom of the table shows the cumulative numbers across all projects. We performed our evaluation on 615 revisions of 32 projects totaling 4,937,189 LOC. To the best of our knowledge, this is the largest dataset used in any RTS study.

We visually separate projects with *short running* and *long running* test suites. While no strict rule defines the boundary between the two, we classified the projects whose test suites execute in less than one minute as short running. The following sections mostly present results for *all* projects together, but in several cases we contrast the results for projects with short- and long-running test suites.

4.2 Experimental Setup

We briefly describe our experimental setup. The goal is to evaluate how EKSTAZI performs if RTS is run for each committed project revision. In general, developers may run RTS even between commits [26], but there is no dataset that would allow *executing* tests the same way that developers executed them in between commits. For each project, our experimental script checks out the revision that is 20 revisions *before* the revision specified in Figure 4. If any re-

	Project	Revision	Buildable Revisions	LOC	Test [*avg*]		Time [sec]		Test Selection [%]		
					classes	methods	*avg*	$\sum$	e%	$t^{\mathcal{AEC}}$	$t^{\mathcal{AE}}$
short running	Cucumber	5df09f85	20	19,939	49	296	8	169	12	99	76
	JodaTime^M	f17223a4	20	82,996	124	4,039	10	214	21	107	75
	Retrofit	810bb53e	20	7,389	15	162	10	217	16	104	90
	CommonsValidator^M	1610469	20	12,171	61	416	11	230	6	88	78
	BVal	1598345	20	17,202	21	231	13	267	13	138	97
	CommonsJXPath^M	1564371	13	24,518	33	386	15	205	20	94	81
	GraphHopper	0e0e311c	20	33,254	80	677	15	303	16	85	59
	River	1520131	19	297,565	14	83	17	335	6	35	18
	Functor	1541713	20	21,688	164	1,134	21	439	13	112	90
	EmpireDB	1562914	20	43,980	23	113	27	546	18	112	99
	JFreeChart	3070	20	140,575	359	2,205	30	618	5	80	64
	CommonsColl4	1567759	20	52,040	145	13,684	32	644	9	66	55
	CommonsLang3	1568639	20	63,425	121	2,492	36	728	11	60	53
	CommonsConfig	1571738	16	55,187	141	2,266	39	633	20	72	58
	PdfBox	1582785	20	109,951	94	892	40	813	12	80	63
	GSCollections	6270110e	20	920,208	1,106	64,614	51	1,036	29	107	90
long running	CommonsNet	1584216	19	25,698	37	215	68	1,300	10	21	21
	ClosureCompiler	65401150	20	211,951	233	8,864	71	1,429	17	62	50
	CommonsDBCP	1573792	16	18,759	27	480	76	1,229	21	46	39
	Log4j^M	1567108	19	30,287	38	440	79	1,508	6	62	43
	JGit^M	bf33a6ee	20	124,436	229	2,223	83	1,663	22	65	50
	CommonsIO	1603493	20	25,981	84	976	98	1,969	12	30	24
	Ivy^M	1558740	18	72,179	121	1,005	170	3,077	38	53	44
	Jenkins (light)	c826a014	20	112,511	86	3,314	171	3,428	7	74	71
	CommonsMath	1573523	20	186,796	461	5,859	249	4,996	6	77	16
	Ant^M	1570454	20	131,864	234	1,667	380	7,613	13	24	21
	Continuum^M	1534878	20	91,113	68	361	453	9,064	10	32	26
	Guava^M	af2232f5	16	257,198	348	641,534	469	7,518	13	45	17
	Camel (core)	f6114d52	20	604,301	2,015	4,975	1,296	25,938	5	9	7
	Jetty	0f70f288	20	282,041	504	4,879	1,363	27,275	26	57	49
	Hadoop (core)	f3043f97	20	787,327	317	2,551	1,415	28,316	7	38	22
	ZooKeeper^M	1605517	19	72,659	127	532	2,565	48,737	20	43	37
$\sum$	-	-	615	4,937,189	7,479	773,565	9,400	182,475	-	-	-
				avg(all)					**14**	**68**	**53**
				avg(short running) | *avg(long running)*					14|**15**	90|**46**	72|**34**

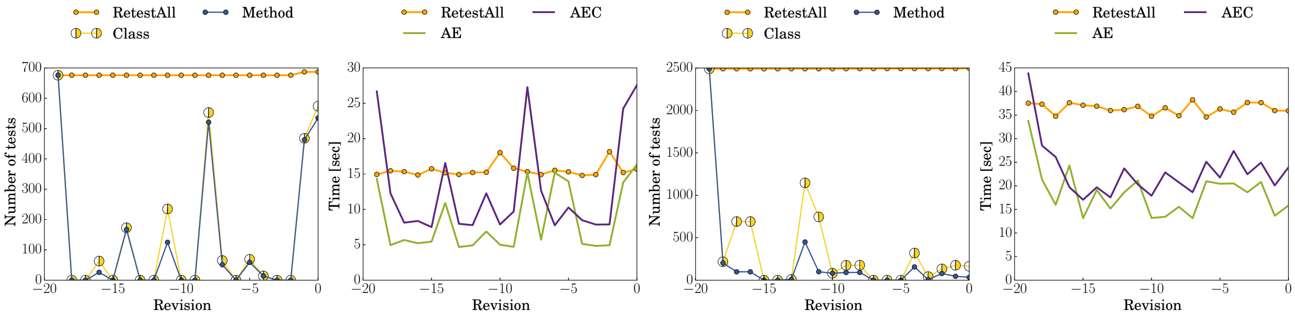
Figure 4: Projects used in the evaluation and test selection results using Ekstazi. Figure 5 describes the column titles

c%	Percentage of test classes selected
e%	Percentage of test entities selected
m%	Percentage of test methods selected
$t^{\mathcal{AEC}}$	Time for $\mathcal{AEC}$ normalized by time for RetestAll
$t^{\mathcal{AE}}$	Time for $\mathcal{AE}$ normalized by time for RetestAll

Figure 5: Legend for symbols used in evaluation

vision cannot build, it is ignored from the experiment, i.e., the analysis phase of the next buildable revision uses dependencies collected at the previous buildable revision. If it can build, the script executes the tests in three scenarios: (1) RetestAll executes all tests without Ekstazi integration, (2) $\mathcal{AEC}$ executes the tests with Ekstazi while collecting dependencies, and (3) $\mathcal{AE}$ executes the tests with Ekstazi but without collecting dependencies, which is preferred in some settings (Section 3.2). The script then repeats these three steps for all revisions until reaching the latest available revision listed in Figure 4.

In each step, the script measures the number of executed tests and the testing time (the execution of all tests for JUnit, the end-to-end time for all $\mathcal{AEC}$ phases of Ekstazi, or just the times for the $\mathcal{AE}$ phases). The script measures the time to *execute the build command* that the developers use to execute the tests (e.g., `ant junit-tests` or `mvn test`). We sometimes limited the tests to just a part of the entire project (e.g., the `core` tests for Hadoop in RetestAll take almost 8 hours across 20 revisions, and the full test suite takes over 17 hours for just one revision). In our evaluation, we did not modify anything in the build configuration file for running the tests, e.g., if it uses multiple cores, excludes some tests, or spawns new JVMs for each test class. By measuring the time for the build command, we evaluate the speedup that the developers would have observed had they used Ekstazi. Note that the speedup that Ekstazi provides over RetestAll is even *bigger* for the testing itself than for the build command, because the build command has some fixed overhead before initiating the testing.



(a) GraphHopper tests (b) GraphHopper time (c) CommonsLang3 tests (d) CommonsLang3 time

Figure 6: Number of available and selected tests (a,c) and end-to-end time in seconds (b,d)

Ekstazi has two main options: selection granularity can be class or method, and smart checksum can be on or off. The default configuration uses the class selection granularity with smart checksum. As discussed earlier, due to idiosyncrasies of JUnit 3, Ekstazi does not run the class selection granularity for all projects; those that use the method selection granularity have the superscript M in Figure 4.

4.3 Main RTS Results

The testing time is the key metric to compare RetestAll, Ekstazi $\mathcal{AEC}$, and Ekstazi $\mathcal{AE}$ runs; as an additional metric, we use the number of executed tests. Figure 6 visualizes these metrics for two of the projects, GraphHopper and CommonsLang3. Plots for other projects look similar; the selected projects include several revisions that are interesting to highlight. For each of the 20 revisions, we plot the total number of test methods (close to 700 and 2,500 in GraphHopper and CommonsLang3, respectively), the number of test methods Ekstazi selects at the method level (blue line), the number of test methods Ekstazi selects at the class level (yellow line), the time for RetestAll (orange line), the time for all $\mathcal{AEC}$ phases of Ekstazi at the method level (purple line), and the time for only $\mathcal{AE}$ phases at the method level (green line). For example, revision −12 for CommonsLang3 has about 400 and 1,200 test methods selected at the method and class level, respectively. We compute the selection ratio for each revision, in this case ∼400/2,500 and ∼1,200/2,500, and then average the ratios over all revisions; for CommonsLang3, the average ratio is about 8% of methods and 11% of classes (Figure 8). Likewise for times, we compute the ratio of the Ekstazi time over the RetestAll time for each revision and then average these ratios. In all starting revisions, −20, we expect the Ekstazi $\mathcal{AEC}$ to be slower than RetestAll, as Ekstazi runs all the tests and collects dependencies. In general, whenever Ekstazi selects (almost) all tests, it is expected to be slower than RetestAll, due to analysis and collection overhead. For example, Ekstazi is slower in revisions −1, −8, and −14 for GraphHopper, but it is faster for many other revisions. We also expect Ekstazi $\mathcal{AE}$ runs to be faster than Ekstazi $\mathcal{AEC}$ runs, but there are some cases where the background processes flip that, e.g., revisions −5 and −6 for GraphHopper. The background noise also makes the time for RetestAll to fluctuate, but over a large number of revisions we expect the noise to cancel out and allow a fair comparison of times for RetestAll, Ekstazi $\mathcal{AEC}$, and Ekstazi $\mathcal{AE}$. The noise has smaller effect on long running test suites.

	All			Smart		
	m%	t$^{\mathcal{AEC}}$	t$^{\mathcal{AE}}$	m%	t$^{\mathcal{AEC}}$	t$^{\mathcal{AE}}$
Camel (core)	5	9	7	5	9	7
CommonsDBCP	40	60	47	23	43	37
CommonsIO	21	42	32	14	30	24
CommonsMath	6	85	16	6	75	17
CommonsNet	11	28	28	9	26	22
CommonsValidator	7	93	79	6	88	78
Ivy	47	63	52	38	53	44
Jenkins (light)	14	72	69	7	74	71
JFreeChart	6	87	70	5	84	67
avg(all)	17	60	45	13	54	41

Figure 7: Ekstazi without and with smart checksums

The last three columns in Figure 4 show the average selection per project; "e%" shows the ratio of test entities (methods or classes) selected, and the times for $\mathcal{AEC}$ and $\mathcal{AE}$ are normalized to the JUnit run without Ekstazi. For example, for Cucumber, Ekstazi selects on average 12% of test entities, but the time that Ekstazi takes is 99% of RetestAll (or 76% if the $\mathcal{C}$ phase is ignored), so it provides almost no benefit. In fact, for some other projects with short-running test suites, Ekstazi is slower than RetestAll; we highlight such cases, e.g., for JodaTime in Figure 4.

Overall, the selection ratio of test entities varies between 5% and 38% of RetestAll, the time for $\mathcal{AEC}$ varies between 9% and 138% (slowdown), and the time for $\mathcal{AE}$ varies between 7% and 99%. On average, across all the projects, the $\mathcal{AEC}$ time is 68%, and the $\mathcal{AE}$ time is 53%. More importantly, all slowdowns are for projects with short-running test suites. Considering only the projects with long-running test suites, Ekstazi reduces the $\mathcal{AEC}$ time to 46% of RetestAll, and reduces the $\mathcal{AE}$ time to 34%. In sum, Ekstazi appears useful for projects whose test suites take over a minute: Ekstazi on average roughly halves their testing time.

4.4 Smart Checksums

Recall that smart checksum performs a more expensive comparison of `.class` files to reduce the number of selected test entities. Figure 7 shows a comparison of Ekstazi runs with smart checksum being off and on, for a diverse subset of projects. While smart checksum improves both the number of selected entities and the testing time (on average and in most cases), there are several cases where the results are the same, or the reduction in the testing time is even slightly lower, e.g., both times for Jenkins or the $\mathcal{AE}$ time for Com-

		Method			Class	
	m%	t$^{\mathcal{AEC}}$	t$^{\mathcal{AE}}$	c%	t$^{\mathcal{AEC}}$	t$^{\mathcal{AE}}$
BVal	16	138	94	13	138	97
ClosureCompiler	20	96	53	17	62	50
CommonsColl4	7	81	60	9	66	55
CommonsConfig	19	76	57	20	72	58
CommonsDBCP	23	43	37	21	46	39
CommonsIO	14	30	24	12	30	24
CommonsLang3	8	63	51	11	60	53
CommonsMath	6	75	17	6	77	16
CommonsNet	9	26	22	10	21	21
Cucumber	13	105	78	12	99	76
EmpireDB	13	117	100	18	112	99
Functor	15	111	100	13	112	90
GraphHopper	19	84	54	16	85	59
GSCollections	16	198	101	29	107	90
JFreeChart	5	84	67	5	80	64
PdfBox	8	85	70	12	80	63
Retrofit	19	113	93	16	104	90
River	6	34	17	6	35	18
avg(*all*)	13	87	61	14	77	59

Figure 8: Ekstazi with both selection granularities

monsMath. This happens if projects have no revision (in the last 20 revisions) that modifies only debug info; using smart checksum then leads to a slowdown as it never selects fewer tests but increases the cost of checking and collecting dependencies. We also manually inspected the results for several projects and found that smart checksum can be further improved: some `.class` files differ only in the order of annotations, but Java specification does not attach semantics to this order, so such changes can be safely ignored. In sum, smart checksum reduces the overall testing time.

4.5 Selection Granularity

Ekstazi provides two levels of selection granularity: methods (which selects fewer tests for the $\mathcal{E}$ phase but makes the $\mathcal{A}$ and $\mathcal{C}$ phases slower) and classes (which makes the $\mathcal{A}$ and $\mathcal{C}$ phases faster but selects more tests for the $\mathcal{E}$ phase). Figure 8 shows a comparison of Ekstazi for these two levels, on several randomly selected projects. Because Ekstazi does not support method selection granularity for projects that use a custom JUnit runner, we do not compare for such projects. Also, we do not compare for Guava; it has a huge number of test methods, and with method selection granularity, our default format for saving dependencies (Section 3.4) would create a huge number of dependency files that may exceed limits set by the file system. The class selection granularity improves both $\mathcal{AEC}$ and $\mathcal{AE}$ times on average and in most cases, especially for GSCollections. In some cases where the class selection granularity is not faster, it is only slightly slower. In sum, the class selection granularity reduces the overall testing time compared to the method selection granularity, and the class selection granularity should be the default value.

4.6 Dependency Granularity

We next evaluate two levels of *dependency granularity*. We compare Ekstazi, which uses the class dependency granularity, with FaultTracer [60], which tracks dependencies on the edges of an extended control-flow graph (ECFG). To the best of our knowledge, FaultTracer was the only publicly available tool for RTS at the time of our experiments.

	FaultTracer		Ekstazi	
	m%	t$^{\mathcal{AEC}}$	m%	t$^{\mathcal{AEC}}$
CommonsConfig	8	223	19	76
CommonsJXPath	14	294	20	94
CommonsLang3	1	183	8	63
CommonsNet	2	57	9	26
CommonsValidator	1	255	6	88
JodaTime	3	663	21	107
avg(*all*)	5	279	14	76

Figure 9: Test selection with FaultTracer and Ekstazi

FaultTracer collects the set of ECFG edges covered during the execution of each *test method*. For comparison purposes, we also use Ekstazi with the method selection granularity. FaultTracer implements a sophisticated change-impact analysis using the Eclipse [35] infrastructure to parse and traverse Java sources of two revisions. Although robust, FaultTracer has several limitations: (1) it requires that the project be an Eclipse project, (2) the project has to have only a single module, (3) FaultTracer does not track dependencies on external files, (4) it requires that both source revisions be available, (5) it does not track reflection calls, (6) it does not select newly added tests, (7) it does not detect any change in the test code, and (8) it cannot ignore changes in annotations that Ekstazi ignores via smart checksum [6]. Due to these limitations, we had to discard most of the projects from the comparison, e.g., 15 projects had multiple modules, and for CommonsIO, FaultTracer was unable to instrument the code.

Figure 9 shows a comparison of FaultTracer and Ekstazi, with the values, as earlier, first normalized to the savings compared to RetestAll for one revision, and then averaged across revisions. The Ekstazi results are the same as in Figure 4 and repeated for easier comparison. The results show that Ekstazi has a much lower end-to-end time than FaultTracer, even though Ekstazi does select more tests to run. Moreover, the results show that FaultTracer is even slower than RetestAll.

To gain confidence in the implementation, we compared the sets of tests selected by Ekstazi and FaultTracer, and confirmed that the results were correct. In most cases, Ekstazi selected a superset of tests selected by FaultTracer. In a few cases, Ekstazi (correctly) selected fewer tests than FaultTracer for two reasons. First, Ekstazi may select fewer tests due to smart checksum (Section 3.5). Second, Ekstazi ignores changes in source code that are not visible at the bytecode level, e.g., local variable rename (Section 3.3).

4.7 Apache CXF Case Study

Several Apache projects integrated Ekstazi into their main repositories. Note that Ekstazi was integrated in these projects after we selected the projects for the evaluation, as explained in Section 4.2. One of the projects that integrated Ekstazi, Apache CXF [12], was not used in our evaluation. To estimate the benefits observed by Apache developers, we performed a study to measure time savings since Ekstazi was integrated (189 revisions at the time of this writing). We measured test execution time, across all revisions, for RetestAll and Ekstazi. Running tests with Ekstazi (∼54h) was 2.74X faster than RetestAll (∼148h).

5. DISCUSSION

Coarser Dependencies Can Be Faster: We argue that the cost of FaultTracer is due to its *approach* and not just due to it being a *research tool.* The cost of FaultTracer stems from collecting fine-grained dependencies, which affects both the $\mathcal{A}$ and $\mathcal{C}$ phases. In particular, the $\mathcal{A}$ phase needs to parse both old and new revisions and compare them. While Orso et al. [45] show how some of that cost can be lowered by filtering classes that did not change, their results show that the overhead of parsing and comparison still ranges from a few seconds up to 4min [45]. Moreover, collecting fine-grained dependencies is also costly. For example, we had to stop FaultTracer from collecting ECFG dependencies of CommonsMath after one hour; it is interesting to note that CommonsMath also has the most expensive $\mathcal{C}$ phase for Ekstazi (~8X for the initial run when there is no prior dependency info). Last but not least, in terms of adoption, FaultTracer and similar approaches are also more challenging than Ekstazi because they require access to the old revision through some integration with version-control systems. In contrast, Ekstazi only needs the checksums of dependent files from the old revision.

Sparse Collection: Although Ekstazi collects dependencies at each revision by default, one can envision collecting dependencies at every n-th revision [16, 18]. Note that this approach is safe as long as the analysis phase checks the changes between the current revision and the latest revision for which dependencies were collected. This avoids the cost of frequent collection but leads to less precise selection.

Duplicate Tests: In a few cases, we observed that Ekstazi did not run some tests during the first run, which initially seemed like a bug in Ekstazi. However, inspecting these cases showed that some test classes were (inefficiently) included multiple times in the same test suite by the original developers. When JUnit (without Ekstazi) runs these classes, they are indeed executed multiple times. But when Ekstazi runs these test suites, after it executes a test for the first time, it saves the test's dependency file, so when the test is encountered again for the same test suite, all its dependent files are the same, and the test is ignored. However, if the same test method *name* is encountered *consecutively* multiple times, Ekstazi does not ignore the non-first runs but unions the dependencies for all those invocations, to support parameterized unit tests.

Parameterized Tests: Recent versions of JUnit support parameterized unit tests [56]. A parameterized test defines a set of input data and invokes a test method with each input from the set; each input may contain multiple values. This approach is used in data-driven scenarios where only the test input changes, but the test method remains the same. Currently, Ekstazi considers a parameterized unit test as a single test and unions the dependencies collected when executing the test method with each element from the input data set. In the future, we could explore tracking individual invocations of parameterized tests.

Flaky Tests: Tests can have non-deterministic executions for multiple reasons such as multi-threaded code, time dependencies, asynchronous calls, etc. If a test passes and fails for the same code revision, it is often called a "flaky test" [39]. Even if a test has the same outcome, it can have different dependencies in different runs. Ekstazi collects dependencies for a *single* run and guarantees that the test will be selected if any of its dependencies changes. However, if a dependency changes for another run that was not observed, the test will not be selected. This is the common approach in RTS [50] because collecting dependencies for all runs (e.g., using software model checking) would be costly.

Parallel Execution vs. RTS: It may be (incorrectly) assumed that RTS is not needed in the presence of parallel execution if sufficient resources are available. However, even companies with an abundance of resources cannot keep up with running all tests for every revision [20, 55, 57]. Additionally, parallel test execution is orthogonal to RTS. Namely, RTS can significantly speed up test execution even if the tests are run in parallel. For example, tests for four projects used in our evaluation (ClosureCompiler, Hadoop, Jenkins, and JGit) execute by default on all available cores. Still, we can observe substantial speedup in testing time when Ekstazi is integrated in these projects. Moreover, RTS itself can be parallelized. In loose integration (Section 3.2), we can run $\mathcal{A}$ of all tests in parallel and then run $\mathcal{EC}$ of all tests in parallel. In tight integration, we can run $\mathcal{AEC}$ of all tests in parallel.

6. THREATS TO VALIDITY

External: The projects used in the evaluation may not be representative. To mitigate this threat, we performed experiments on a number of projects that vary in size, number of developers, number of revisions, and application domain.

We performed experiments on 20 revisions per project; the results could differ if we selected more revisions or different segments from the software history. We considered only 20 revisions to limit machine time needed for experiments. Further, we consider each segment right before the latest available revision at the time when we started the experiments on the project.

The reported results for each project were obtained on a single machine. The results may differ based on the configuration (e.g., available memory). We also tried a small subset of experiments on another machine and observed similar results in terms of speedup, although the absolute times differed due to machine configurations. Because our goal is to compare real time, we did not want to merge experimental results from different machines.

Internal: Ekstazi implementation may contain bugs that may impact our conclusions. To increase the confidence in our implementation, we reviewed the code, tested it on a number of small examples, and manually inspected several results for both small and large projects.

Construct: Although many RTS techniques have been proposed, we compared Ekstazi only with FaultTracer. To the best of our knowledge, FaultTracer was the only publicly available RTS tool. Our focus in comparison is not only on the number of selected tests but primarily on the end-to-end time taken for testing. We believe that the time that the developer observes, from initiating the test-suite execution for the new code revision until all the test outcomes become available, is the most relevant metric for RTS.

7. RELATED WORK

There has been a lot of work on regression testing in general, as surveyed in two reviews [14, 59], and on RTS in particular, as further surveyed in two more reviews [22, 23]. We first discuss work that inspired our Ekstazi technique: recent advances in build systems [2, 3, 7–9, 17, 24, 42], prior work on RTS based on class dependencies [21, 34, 36, 45, 51, 52], and external resources [29, 30, 43, 58]. We then discuss prior techniques that use different levels of dependency granularity and other related work.

Build Systems and Memoization: Memoize [42], which is a Python-based system, uses `strace` on Linux to monitor all files opened while the given command executes. Memoize saves all file paths and file checksums, and ignores subsequent runs of the same command if no checksum changed. Fabricate [24] is an improved version of Memoize that also supports parallel builds. Other build systems, for example SCons [8] and Vesta [9], capture dependencies on files that are attempted to be accessed, even if they do not exist. For memoization of Python code, Guo and Engler proposed IncPy [28] that memoizes calls to functions. IncPy supports functions that use files, i.e., it stores the file checksums and re-executes a function if any of its inputs or files change. Our insight is to view RTS as memoization: if none of the dependent files for some test changed, then the test need not be run. By capturing file dependencies for each test entity, Ekstazi provides scalable and efficient RTS that integrates well with testing frameworks, increasing the chance of adoption. Ekstazi differs from build systems and memoization in several aspects: capturing dependencies for each test entity even when all entities are executed in the same JVM, supporting test entity granularities, smart checksums, and capturing files inside archives (i.e., classfiles inside jar files).

Class-based Test Selection: Hsia et al. [34] were the first to propose RTS based on class firewall [36], i.e., the statically computed set of classes that may be affected by a change. Orso et al. [45] present an RTS technique that combines class firewall and dangerous edges [50]. Their approach works in two phases: it finds relations between classes and interfaces to identify a subgraph of the Java Interclass Graph that may be affected by the changes, and then selects tests via an edge-level RTS on the identified subgraph. Skoglund and Runeson first performed a large case study on class firewall [51] and then [52] proposed an improved technique that removes the class firewall and uses a change-based RTS technique that selects only tests that execute modified classes. More recently, Christakis et al. [17] give a machine-verifiable proof that memoization of partial builds is safe when capturing dependencies on all files, assuming that code behaves deterministically (e.g., there is no network access). Compared to prior work, Ekstazi captures all files (including classes) for RTS, handles addition and changes of test classes, applies smart checksums, supports reflection, and has both class and method selection granularity. Moreover, we integrated Ekstazi with JUnit and evaluated on a much larger set of projects, using the end-to-end testing time.

External Resources: Haraty et al. [29] and Daou [30] explored regression testing for database programs. Willmor and Embury [58] proposed two RTS techniques, one that captures interaction between a database and the application, and the other based solely on the database state. Nanda et al. [43] proposed RTS for applications with configuration files and databases. We explored several ways to integrate RTS with the existing testing frameworks, and Ekstazi captures all files. At the moment, Ekstazi offers no special support for dependencies other than files (e.g., databases and web services). We plan to support these cases in the future.

Granularity Levels: Ren et al. [48] described the Chianti approach for change impact analysis. Chianti collects method dependencies for each test and analyzes differences at the source level. We show that fine-grained analysis can be expensive. Echelon from Microsoft [54] performs test prioritization [59] rather than RTS. It tracks fine-grained dependencies based on basic blocks and accurately computes changes between code revisions by analyzing compiled binaries. Most research RTS techniques [59] also compute fine-grained dependencies like Echelon. Because Echelon is not publicly available, our evaluation used FaultTracer [60], a state-of-the-research RTS tool. Elbaum et al. [19] and Di Nardo et al. [44] compared different granularity levels (e.g., statement vs. function) for test prioritization techniques. Ekstazi uses a coarse granularity, i.e., files, for dependency granularity, and the experiments show better results than for FaultTracer based on a finer granularity. Also, the comparison of both selection granularities shows that the coarser granularity provides more savings.

Other Related Work: Zheng et al. [61] propose a fully static test selection technique that does not collect dependencies but constructs a call graph (for each test) and intersects the graph with the changes. Ekstazi collects dependencies dynamically, and measures the end-to-end time. Pinto et al. [46] and Marinescu et al. [41] studied test-suite evolution and other execution metrics over several project revisions. While we did not use those same projects and revisions (e.g., Marinescu et al.'s projects are in C), we did use 615 revisions of 32 projects. Several prediction models [32, 49] were proposed to estimate if RTS would be cheaper than RetestAll. Most models assume that the $\mathcal{C}$ phase is run separately and would need to be adjusted when RTS runs all phases and the end-to-end time matters. We focus Ekstazi on the end-to-end time.

8. CONCLUSIONS

We described the Ekstazi technique and its implementation for regression test selection. The key difference between Ekstazi and prior work is that Ekstazi tracks test dependencies on files and integrates with JUnit. For each test entity, Ekstazi collects a set of files that are accessed during the execution. Ekstazi detects affected tests by checking if the dependent files changed.

We aim to make RTS practical by balancing the time for the analysis and collection phases, rather than focusing solely on reducing the number of selected tests for the execution phase. The use of coarse-grain dependencies provides a "sweet-spot". The experiments show that Ekstazi already performs better overall than the state-of-the-research tools. Most importantly, Ekstazi works faster than the RetestAll base case. Finally, Ekstazi has already been integrated in the main repositories by several popular open source projects, including Apache Camel, Apache Commons Math, and Apache CXF.

9. ACKNOWLEDGMENTS

We thank Alex Gyori, Farah Hariri, Owolabi Legunsen, Jessy Li, Yu Lin, Qingzhou Luo, Aleksandar Milicevic, and August Shi for their feedback on this work; Nikhil Unni, Dan Schweikert, and Rohan Sehgal for helping to make the list of projects used in the evaluation; and Lingming Zhang for making FaultTracer available. This research was partially supported by the NSF Grant Nos. CNS-0958199, CCF-1012759, CCF-1421503, and CCF-1439957.

10. REFERENCES

[1] Apache Projects. https://projects.apache.org/.

[2] Buck. http://facebook.github.io/buck/.

[3] Build in the cloud. http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html.

[4] GitHub. https://github.com/.

[5] Google Code. https://code.google.com/.

[6] Java Annotations. http://docs.oracle.com/javase/7/docs/technotes/guides/language/annotations.html.

[7] Ninja. https://martine.github.io/ninja/.

[8] SCons. http://www.scons.org/.

[9] Vesta. http://www.vestasys.org/.

[10] Apache Camel - Building. http://camel.apache.org/building.html.

[11] Apache Commons Math. https://github.com/apache/commons-math.

[12] Apache CXF. https://github.com/apache/cxf.

[13] B. Beizer. *Software Testing Techniques (2nd Ed.)*. 1990.

[14] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.

[15] P. K. Chittimalli and M. J. Harrold. Re-computing coverage information to assist regression testing. In *International Conference on Software Maintenance*, pages 164–173, 2007.

[16] P. K. Chittimalli and M. J. Harrold. Recomputing coverage information to assist regression testing. *Transactions on Software Engineering*, 35(4):452–469, 2009.

[17] M. Christakis, K. R. M. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *International Symposium on Formal Methods*, pages 643–657, 2014.

[18] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *International Conference on Software Maintenance*, pages 170–179, 2001.

[19] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Transactions on Software Engineering*, 28(2):159–182, 2002.

[20] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[21] E. Engström and P. Runeson. A qualitative survey of regression testing practices. In *Product-Focused Software Process Improvement*, volume 6156, pages 3–16. Springer-Verlag, 2010.

[22] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information & Software Technology*, 52(1):14–30, 2010.

[23] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *International Symposium on Empirical Software Engineering and Measurement*, pages 22–31, 2008.

[24] Fabricate. https://code.google.com/p/fabricate/.

[25] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *International Conference on Software Engineering, demo papers*, 2015. To appear.

[26] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Automated Software Engineering*, pages 361–372, 2014.

[27] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[28] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *International Symposium on Software Testing and Analysis*, pages 287–297, 2011.

[29] R. A. Haraty, N. Mansour, and B. Daou. Regression testing of database applications. In *Symposium on Applied Computing*, pages 285–289, 2001.

[30] R. A. Haraty, N. Mansour, and B. Daou. Regression test selection for database applications. *Advanced Topics in Database Research*, 3:141–165, 2004.

[31] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, 2001.

[32] M. J. Harrold, D. S. Rosenblum, G. Rothermel, and E. J. Weyuker. Empirical studies of a prediction model for regression test selection. *Transactions on Software Engineering*, 27(3):248–263, 2001.

[33] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *International Conference on Software Maintenance*, pages 362–367, 1988.

[34] P. Hsia, X. Li, D. Chenho Kung, C.-T. Hsu, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *Journal of Software Maintenance: Research and Practice*, 9(4):217–233, 1997.

[35] Eclipse Kepler. http://www.eclipse.org/kepler/.

[36] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[37] H. Leung and L. White. A cost model to compare regression test strategies. In *International Conference on Software Maintenance*, pages 201–208, 1991.

[38] H. K. N. Leung and L. White. Insights into regression testing. In *International Conference on Software Maintenance*, pages 60–69, 1989.

[39] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[40] A. G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *International Conference on Software Maintenance*, pages 204–213, 2002.

[41] P. D. Marinescu, P. Hosek, and C. Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *International Symposium on Software Testing and Analysis*, pages 93–104, 2014.

[42] Memoize. `https://github.com/kgaughan/memoize.py`.

[43] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *International Conference on Software Testing, Verification, and Validation*, pages 21–30, 2011.

[44] D. D. Nardo, N. Alshahwan, L. C. Briand, and Y. Labiche. Coverage-based test case prioritisation: An industrial case study. In *International Conference on Software Testing, Verification and Validation*, pages 302–311, 2013.

[45] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*, pages 241–251, 2004.

[46] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[47] PIT. `http://pitest.org/`.

[48] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.

[49] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *Transactions on Software Engineering*, 23(3):146–156, 1997.

[50] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[51] M. Skoglund and P. Runeson. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *International Symposium on Empirical Software Engineering*, pages 74–83, 2005.

[52] M. Skoglund and P. Runeson. Improving class firewall regression test selection by removing the class firewall. *International Journal of Software Engineering and Knowledge Engineering*, 17(3):359–378, 2007.

[53] sloccount. `http://www.dwheeler.com/sloccount/`.

[54] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *Software Engineering Notes*, 27(4):97–106, 2002.

[55] Testing at the speed and scale of Google, Jun 2011. `http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`.

[56] N. Tillmann and W. Schulte. Parameterized unit tests. In *International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[57] Tools for continuous integration at Google scale, October 2011. `http://www.youtube.com/watch?v=b52aXZ2yi08`.

[58] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *International Conference on Software Maintenance*, pages 421–430, 2005.

[59] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[60] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *International Conference on Software Maintenance*, pages 23–32, 2011.

[61] J. Zheng, B. Robinson, L. Williams, and K. Smiley. An initial study of a lightweight process for change identification and regression test selection when source code is not available. In *International Symposium on Software Reliability Engineering*, pages 225–234, 2005.

[62] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for COTS-based applications. In *International Conference on Software Engineering*, pages 512–522, 2006.