Efficient Dependency Detection for Safe Java Test Acceleration

Jonathan Bell, Gail Kaiser
Columbia University
500 West 120th St
New York, NY USA
{jbell,kaiser}@cs.columbia.edu

Eric Melski, Mohan Dattatreya
Electric Cloud, Inc
35 S Market Street
San Jose, CA USA
{ericm,mohan}@electric-cloud.com

ABSTRACT

Slow builds remain a plague for software developers. The frequency with which code can be built (compiled, tested and packaged) directly impacts the productivity of developers: longer build times mean a longer wait before determining if a change to the application being built was successful. We have discovered that in the case of some languages, such as Java, the majority of build time is spent running tests, where dependencies between individual tests are complicated to discover, making many existing test acceleration techniques unsound to deploy in practice. Without knowledge of which tests are dependent on others, we cannot safely parallelize the execution of the tests, nor can we perform incremental testing (i.e., execute only a subset of an application's tests for each build). The previous techniques for detecting these dependencies did not scale to large test suites: given a test suite that normally ran in two hours, the best-case running scenario for the previous tool would have taken over 422 CPU days to find dependencies between all test methods (and would not soundly find all dependencies) — on the same project the exhaustive technique (to find all dependencies) would have taken over 10^{300} years. We present a novel approach to detecting all dependencies between test cases in large projects that can enable safe exploitation of parallelism and test selection with a modest analysis cost.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Test dependence, detection algorithms, empirical studies

1. INTRODUCTION

Slow builds remain a hinderance to continuous integration and deployment processes, with the majority of building time often spent in the testing phase. Our industry partners confirm previous results reported in literature [35]: test suites frequently take several hours to run — often over a day — making it hard to run them with the desired frequency. Our study of popular open source Java programs echoes these results, finding projects that take hours to build, with most of that time spent testing. Even in cases of projects that build in a manageable amount of time — for example, five to ten minutes — faster builds can result in a significant increase in productivity due to less lag-time for test results.

To make testing faster, developers may turn to techniques such as Test Suite Minimization (which reduce the size of a test suite, for instance by removing tests that duplicate others) [11, 12, 23, 24, 27, 28, 37, 39], Test Suite Prioritization (which reorders tests to run those most relevant to recent changes first) [14, 15, 35, 36, 38], or Test Selection [17, 25, 32] (which selects tests to execute that are impacted by recent changes). Alternatively, given a sufficient quantity of cheap computational resources (e.g. Amazon's EC2), we might hope that we could reduce the amount of wall time needed to run a given test suite even further by parallelizing it.

All of these techniques involve executing tests out of order (compared to their typical execution — which may be random but is almost always alphabetically), making the assumption that individual test cases are *independent*. If some test case t_1 writes to some persistent state, and t_2 depends on that state to execute properly, we would be unable to safely apply previous work in test parallelization, selection, minimization, or prioritization without knowledge of this dependency. Previous work by Zhang et al. has found that these dependencies often come as a surprise and can cause unpredictable results when using common test prioritization algorithms [40].

This assumption is part of the *controlled regression testing assumption*: given a program P and new version P', when P' is tested with test case t, all factors that may influence the outcome of this test (except for the modified code in P') remain constant [34]. This assumption is key to maintaining the soundness of techniques that reorder or remove tests from a suite. In the case of test dependence, we specifically assume that by executing only some tests, or executing them in a different order, we are not effecting their outcome (i.e., that they are independent).

One simple approach to accelerating these test suites is to ignore these dependencies, or hope that developers specify them manually. However, previous work has shown that inadvertently dependent tests exist in real projects, can take significant time to identify, and pose a threat to test suite correctness when applying test acceleration techniques [29, 40]. Zhang et al. show that dependent tests are a serious problem, finding in a study of five open source applications 96 tests that depend on other tests [40]. In our own study we

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08
http://dx.doi.org/10.1145/2786805.2786823

found many large test suites in popular open source software do not isolate their tests, and hence, may potentially have dependencies.

Our new approach and tool, *ElectricTest*, detects dependencies between test cases in both small and large, real-world test suites.*ElectricTest* monitors test execution, detecting dependencies between tests, adding on average a 20x slowdown to test execution when soundly detecting dependencies. In comparison, we found that the previous state of the art approach applied to these same projects showed an average slowdown of 2,276x (using an unsound heuristic not guaranteed to find all dependencies), often requiring more than 10^{308} times the amount of time needed to run the test suite normally in order to exhaustively find all dependencies. Moreover, the existing technique does not point developers to the specific code causing dependencies, making inspection and analysis of these dependencies costly.

With *ElectricTest*, it becomes feasible to soundly perform test parallelization and selection on large test suites. Rather than detect *manifest dependencies* (i.e., a dependency that changes the outcome of a test case, the definition in previous work by Zhang et al, DTDetector [40]), *ElectricTest* detects simple data dependencies and anti-dependencies (i.e., read-over-write and write-over-read). Since not all data dependencies will result in manifest dependencies, our approach is inherently less precise than DTDetector at reporting "true" dependencies between tests, though it will never miss a dependency that DTDetector would have detected. However, in the case of long running test suites (e.g. over one hour), the DTDetector approach is not feasible. On popular open source software, we found that the number and type of dependencies reported by *ElectricTest* allow for up to 16X speedups in test parallelization.

Our key insight is that, for memory-managed languages, we can efficiently detect data dependencies between tests by leveraging existing efficient heap traversal mechanisms like those used by garbage collectors, combined with filesystem and network monitoring. For *ElectricTest*, test T_2 depends on test T_1 if T_2 reads some data that was last written by T_1. A system that logs all data dependencies will always report at least as many dependencies as a system that searches for manifest dependencies. Our approach also provides additional benefits to developers: it can report the exact line of code (with stack trace) that causes a dependency between tests, greatly simplifying test debugging and analysis.

ElectricTest instruments all classes in the system under test (including those provided in the core Java system library) to support a heap-walking analysis. During test execution, *ElectricTest* observes the heap values, files, and network resources that are read and written, collecting these results and analyzing them to determine if a dependency occurred. After this learning phase, we can safely perform test parallelization or selection using the resulting dependency chains produced by *ElectricTest*.

We applied *ElectricTest* to the test suites of 10 popular free open source applications, studying the dependencies detected and the runtime overhead imposed by the detection process. We found that *ElectricTest* finds at least as many dependencies as the state-of-the-art tool in many orders of magnitude less time. We have studied the impact of *ElectricTest* on test parallelization and test selection techniques, finding that its test dependence analysis technique can allow for sound test acceleration. While we implement *ElectricTest* in Java, we believe that we present a sufficiently general approach that should be applicable to other memory-managed languages.

2. MOTIVATION

To motivate our work, we set out to answer three motivating questions to ground our approach:

MQ1: For those projects that take a long time to build, what component of the build dominates that time?

MQ2: Are existing test acceleration approaches safe to apply to real world, long running test suites?

MQ3: Can the state of the art in test dependency detection be practically used to safely apply test acceleration to these long running test suites?

2.1 A Study of Java Build Times

In our previous work [8], we studied 20 open source Java applications to determine the relative amount of build time spent testing, finding testing to consume on average 78% of build time. The longest of these projects took approximately 40 minutes to build, while the shortest completed in under one minute. Given a desire to target projects with very long build times, we wanted to make sure that those very long running builds were also spending most of their time in tests. If we are sure that most of the time spent building these projects is in the testing phase, then we can be confident that a reduction in testing time will have a strong impact in reducing overall build time.

For this study, we downloaded the 1,966 largest and most popular Java projects from the open source repository site, GitHub (those 1,000 with the most forks and stars overall, and those 1,000 with the most forks over 300 MB, as of December 23rd, 2014). From these projects, we searched for only those with tests (i.e., had files that had the word "test" in their name), bringing our list to 921 projects.

Next, we looked at the different build management systems used by each project: there are several popular build systems for Java, such as ant, maven, and gradle. To measure the per-step timing of building each of these projects, we had to instrument the build system, and hence, we selected the most commonly used system in this dataset. We looked for build files for five build systems: ant, maven, gradle, set, and regular Makefiles. Of these 921 projects, the majority (599) used maven, and hence, we focused our study on only those projects using maven due to resource limitations creating and running experiments.

We utilized Amazon's EC2 "m3.medium" instances, each running Ubuntu 14.04.1 and Maven 3.2.5 with 3.75GB of RAM, 14 GB of SSD disk space, and a one-core 2.5Ghz Xeon processor. We tried to build each project first with Java 1.8.0_40, and then fell back to Java 1.7.0_60 if the newer version did not work (some projects required the latest version while others didn't support it). For each project, we first built it in its entirety without any instrumentation, and then we built it again from a clean checkout with our instrumented version of Maven in "offline" mode (with external dependencies already downloaded and cached locally).

If a project contained multiple maven build files, we executed maven on the build file nearest the root of the repository, and we did not perform any per-project configuration. Of these 599 projects, we could successfully build 351.

Table 1 shows the three longest build phases, first for all of these projects, and then filtering to only those projects that took more than 10 minutes to build (69 projects), and

Table 1: Top three phases in Java builds.

Phase	All	Only projects building in:	
	Projects	>10 min	>1 hour
Test	41.22%	59.64%	90.04%
Compile	38.33%	26.25%	8.46%
Package	15.49%		1.05%
Pre-Test		13.51%	

those that took more than one hour to build (8 projects). When looking across all projects, 41% of the build time (per project) was spent testing, and testing was the single most time consuming build step. When eliminating the cases of projects with particularly short build times (those taking less than 10 minutes to execute all phases of the build), the average testing time increased significantly to nearly 60%. In the eight cases of projects that took more than an hour to build, nearly all time (90%) is spent testing. Therefore, to answer **MQ1**, we find that testing dominates build times, especially in long running builds. This conclusion underscores the importance of accelerating testing.

2.2 Danger of Dependent Tests

Any test acceleration technique that executes only a subset of tests, or executes them out of order (e.g., test parallelization or test selection) is unsound in the presence of test dependencies. If the result of one test depends on the execution of a previous test, then these techniques may cause false positives (tests that should fail but pass) or false negatives (tests that should pass but fail).

Zhang et al. studied the issue trackers of five popular open source applications to determine if dependent tests truly exist and cause problems for developers [40]. They found a total of 96 dependent tests, 95 of which would result in a false negative when executed out of order (causing a test to fail although it should pass), and one which produced a false positive when executed out of order (causing a test to pass when it should fail). Given that test dependencies exist and can cause tests to behave incorrectly when executed out of order, we conclude that yes: dependent tests pose a risk to existing test acceleration techniques.

If we isolate the execution of each of our test cases, then dependencies would not be possible. In practice, tests are typically written as single test methods, which are grouped into test classes, which are batched together into modules. Typically each test method represents an atomic test, while test classes represent groups of tests that test the same component. The module separation occurs when a project is split into modules, with a test suite for each module.

Since they are typically testing the same component, individual test methods are never isolated, although sometimes test classes are isolated. Since they represent different modules of code (that must compile separately), test modules are always isolated in our experience. We are interested in detecting dependencies both at the level of individual test methods, and also test classes, which also are the same granularity used by test selection and parallelization techniques. For the remainder of this paper, when we refer to individual tests, we will refer to test classes and test modules.

One approach to solving the dependent test problem is to simply isolate each test to ensure that no dependencies could occur (e.g., by executing each test in its own process, or by using our efficient isolation system VmVm [7]). However, if the application does not isolate its tests, and tests currently depend on each other, then tests may present false negatives or false positives (albeit deterministically between executions) when isolated.

We examined the 351 Java projects that we built, finding that 18 (or 5%) isolated all of their test classes, and 41 (or 12%) isolated at least some of their test classes (i.e., some classes were isolated and others were grouped together and executed without isolation). The majority of projects did not isolate their tests at all, and therefore are prone to test dependencies occurring, posing a risk to test acceleration.

This result differs from our 2013 study, which showed 41% of 591 Java projects isolated their tests [7]. This study examined only projects that built with maven, while our previous study (which was performed through a static analysis of build scripts) examined both maven and ant-building projects. In our previous study, we found that of our 591 projects, only approximately 10% of those that used maven to build and run their tests isolated some or all of their tests, a number much more similar to what we found here.

Due to the risks that they impose and ability to occur (when tests aren't isolated), our goal is to detect dependencies between test classes so that we can (1) inform existing test acceleration techniques of the dependencies to ensure sound acceleration, and (2) provide feedback to developers so that they are aware of dependencies that exist.

2.3 Feasibility of Existing Approaches

Finally, we study the existing state-of-the-art approach for detecting dependencies between test cases to determine if it is feasible to apply to long-running test suites.

If we define a test dependence as the case where executing some set of tests T in a different order changes the result of the test(s), then identifying test dependencies is NP-Complete [40]. This definition for dependence (henceforth referred to as a *manifest dependence*) is more narrow than ours (a distinction described later in §3), but is the definition used in the state-of-the-art work by Zhang et al. [40].

To identify all manifest test dependencies in a suite of n tests we would have to execute every permutation of those n tests, requiring $O(n!)$ test executions, clearly infeasible for any reasonably large test suite. Moreover, such a technique would only identify that tests are dependent, and not the specific resource or lines of code causing the dependence, making it difficult for developers who wish to examine or remove the dependency. In our study that follows, we estimated that this exhaustive process often would take more than 1×10^{308} times longer than running the test suite normally. Zhang et al. propose two techniques to reduce the number of test executions needed to detect manifest dependent tests, both of which they acknowledge may not scale to large test suites [40].

In one approach, they reduce the search space to $O(n^2)$ by suggesting that most dependencies manifest between only two tests, with no need to consider every possible n size permutation. However, this is incomplete: there may be dependencies that only manifest when more than two tests interact. They further reduce the search space by a constant factor (it is still an $O(n^2)$ algorithm) by only checking test combinations that share common resources (defined to be static fields and files). If two tests access (read or write) the same file or static field, then they are marked as sharing a common resource, regardless of whether a true data dependency exists or not. Since this very coarse dependency

Table 2: Testing time and statistics for the 10 longest-running test suites studied with unisolated tests, plus the 4 projects studied in previous work by Zhang et al. [40]. In addition to the normal testing time, we estimate the time that needed to run all pairwise combinations of tests, and the time needed to exhaustively run all combinations.* indicates a slowdown greater than 1×10^{308}.

	Project	Test Classes	Test Methods	Testing Time (mins)	Pairwise Test Slowdown		Exhaustive Test Slowdown	
					Class	Method	Class	Method
Projects selected in §2.3	camel	5,919	13,562	109.70	1,865X	8,045X	*1E+308X	*1E+308X
	crunch	62	243	17.58	65X	298X	54E+82X	54E+82X
	hazelcast	297	2,623	47.37	147X	2,536X	*1E+308X	*1E+308X
	jetty.project	554	5,603	20.08	35X	2,555X	2E+60X	2E+60X
	mongo-java-driver	58	576	74.25	58X	649X	4E+76X	4E+76X
	mule	2,047	10,476	117.45	250X	3,438X	*1E+308X	*1E+308X
	netty	289	4,601	62.95	11X	2,725X	62E+82X	62E+82X
	spring-data-mongodb	141	1,453	121.38	136X	1,715X	3E+230X	3E+230X
	tachyon	53	362	34.47	47X	397X	56E+56X	56E+56X
	titan	177	1,191	81.82	181X	398X	*1E+308X	*1E+308X
	Average	960	4,069	68.71	279X	2,276X	*1E+308X	*1E+308X
Zhang [40]	joda-time	122	3,875	0.27	627X	418,016X	3E+204X	*1E+308X
	xml security	19	108	0.37	59X	1,316X	47E+16X	15E+172X
	crystal	11	75	0.07	37X	763X	3E+8X	3E+108X
	synoptic	27	118	0.03	183X	3,497X	5E+28X	1E+194X
	Average	45	1,044	0.18	226X	105,898X	70E+202X	*1E+308X

detection will likely result in many false positives, Zhang et al. manually inspect each resource to determine if it is likely to cause a dependence, and if not, ignore it in this process. This heuristic can limit the search space but it still can remain large, and requires manual effort to rule out some resource accesses that will not cause manifest dependencies.

Table 2 shows the estimated CPU time needed to detect the dependent tests in each of the ten longest building projects from our dataset from §2.1 with unisolated tests, along with the four projects studied by Zhang et al. previously [40]. This experiment was performed on Amazon EC2 "r3.xlarge" instances, each running Ubuntu 14.04.1 and Maven 3.2.5 with 4 virtualized Intel Xeon X5-2670 v2 2.5Ghz CPUs, 30.5 GB of RAM and 80 GB of SSD storage. Subjects 'jetty', 'titan' and 'crunch' were evaluated on OpenJDK Java 1.7.0_60 (the most recent version supported by the projects) while the others were evaluated on OpenJDK Java 1.8.0_40.

In the case of the four small projects previously studied by Zhang et al, we used their publicly available tool to calculate the pairwise testing time, and estimated the exhaustive testing time. In the case of our ten projects, we estimate all times, due to scaling limitations of the DTDetector tool. We estimated all times using the following approach: first we measured the time to run each test normally and then we calculated the permutations of tests to run for each module of each project (most of these projects had many modules with tests, and since tests from different modules were isolated, there was no need to include permutations cross-module). We added a constant time of 1 second to each combination of tests executed to account for the time needed to start and stop the JVM and system under test (a conservative estimate based on our prior results [7]).

We compare this projected time to the actual time needed to run the test suite in its normal configuration, presenting the slowdown as $T_{DTDetector}/T_{normal}$. Even the pairwise heuristic (examining only every 2-pair of tests, rather than all possible permutations) can be cost prohibitive: adding an overhead of up to 418,016X (minimum 298X for test methods), even though there is no guarantee of its correctness. A large slowdown appears in both long-building and fast building projects.

For the four projects previously studied by Zhang et al., the dependence-aware approach showed approximately one order of magnitude less overhead. However, we were unable to evaluate the dependence-aware technique on our ten projects due to technical limitations of the DTDetector implementation: running it requires manual enumeration and configuration of each test to run in the DTDetector test runner. Given the manual effort required and that this heuristic is unsound, we chose not to implement it for our ten projects.

As expected, there is no situation in the projects that we studied where the fully exhaustive method (testing all possible permutations) is feasible. Even in the cases of the more modest length test suites, the overhead of DTDetector is very high. We answer MQ3 and conclude that the existing, state of the art approach for detecting dependencies between tests can not scale to detect dependencies in the wild, except when using unsound heuristics on the very smallest of test suites that took less than a minute to execute normally.

3. DETECTING TEST DEPENDENCIES

While previous work in the area has focused on detecting *manifest dependencies* between tests [40], we focus instead on a more general definition of dependence. For our purposes, if T_2 reads some value that was last written by T_1, then we say that T_2 depends on T_1 (i.e., there is a data dependence). If some later test, T_3 writes over that same data, then we say that there is an anti-dependence between tests T_2 and T_3: T_3 must never run between T_1 and T_2. Note that any two tests that are *manifest dependent* will also be dependent by our definition, but two tests that have a data dependence may not have a manifest dependence.

Consider the case of a simple utility function that caches the current formatted timestamp at the resolution of seconds so that multiple invocations of the method in the same second returns the same formatted string. If the date formatter has no side-effects we can surmise that if several tests call

this method, while there is a data dependency between them (since the cache is reused), this dependence won't in and of itself influence the outcome of any tests. Hence, there will be no manifest dependence between these tests even though there is a data dependence.

While detecting *manifest* dependencies between tests may require executing every possible permutation of all tests, detecting data dependencies (that may or may not result in manifest dependencies) requires that each test is executed only once. *ElectricTest* detects dependencies by observing global resources read and written by each test, and reports any test T_j that reads a value last written by test T_i as dependent. *ElectricTest* also reports anti-dependencies, that is, other tests T_k that write that same data after T_j, to ensure that T_k is not executed between T_i and T_j.

ElectricTest consists of a static analyzer/instrumenter and a runtime library. Before tests are run with *ElectricTest*, all classes in the system under test (including its libraries) are instrumented with heap tracking code (at the bytecode level — no access to source code is required). In principle, this instrumentation could occur on-the-fly during testing as classes are loaded into JVM, however, we perform the instrumentation offline for increased performance, as many external library classes may remain constant between different versions of the same project. This process is fairly fast though: analyzing and instrumenting the 67,893 classes in the Java 1.8 JDK took approximately 4 minutes on our commodity server. *ElectricTest* detects dynamically generated classes that are loaded when testing (which were not statically instrumented) and instruments them on the fly. During test execution, the *ElectricTest* runtime monitors heap accesses to detect dependencies between tests.

Dependencies between tests can arise due to shared memory, shared files on a filesystem, or shared external resources (e.g. on a network). *ElectricTest*'s approach for file and network dependency detection is simple: it maintains a list of files and network socket addresses that are read and written during each test. *ElectricTest* leverages Java's built in *IOTrace* support to track file and network access. Efficiently detecting in-memory dependencies is much more complex, and we focus our discussion to this technique next.

3.1 Detecting In-Memory Dependencies

To detect dependencies in memory between test cases, *ElectricTest* carefully examines reads and writes to heap memory. Recall that Java is a memory managed language, where it is impossible to directly address memory. Simply put, the heap can be accessed through pointers to it that already exist on the stack, or via `static` fields (which reside in the heap and can be directly referenced).

At the start of each test, we'll assume that the test runner (which is creating these tests) does not pass (on the stack) references to heap objects, or at least not the same reference to multiple tests. We easily verified this safe assumption, as there is typically only a single test runner that's shared between all projects using that framework (e.g. JUnit, which creates a new instance of each test class for each test).

Therefore, our possible leakage points for dependencies between tests will arise through `static` fields. `static` fields are heap roots: they are directly accessed, and therefore the level of granularity at which we detect dependencies.

Unfortunately, to soundly detect all possible dependencies, it is insufficient to simply record accesses to `static` fields, since each `static` field may in turn point to some object which has other `instance` fields. If we simply recorded only accesses to `static` fields (as our previous work, VmVm did [7]), we wouldn't be able to detect all data dependencies, since we wouldn't be able to follow all pointers. With VmVm, we were forced to treat all static field reads as writes, since a test might read a static field to get a pointer to some other part of the heap, then write that other part (indirectly writing the area referenced by the static field).

Our key insight is that we can efficiently detect these dependencies by leveraging several powerful features that already exist in the JVM: garbage collection and profiling. The high level approach that *ElectricTest* uses to efficiently detect in-memory dependencies between test cases is twofold. At the end of each test execution, we force a garbage collection and mark any reachable objects (that weren't marked as written yet) as written in this test case. During the following test executions, we monitor all accesses to the marked objects, and if a test reads an object that was written during a previous test, we tag it as having a dependence on the last test that wrote that object. This method is similarly used to detect anti-dependencies (write after read).

ElectricTest heavily leverages the JVM Tooling Interface (JVMTI), which provides support for internal monitoring and is used for implementing profilers and debuggers that interact with the JVM [31]. While it would be possible (and likely more performant) to implement *ElectricTest* by modifying certain aspects of the JVM directly (e.g. to piggy-back generation counters already used for garbage collection to track which test wrote an object), we chose instead to use the standard JVMTI interface so that *ElectricTest* will not require a specialized JVM: it functions on commodity JVMs such as Oracle's HotSpot or OpenJDK's IcedTea.

Aside from bytecode instrumentation, *ElectricTest* utilizes three key functions of the JVMTI API: heap walking, heap tagging, and heap access notifications. The heap walking mechanism provides a fairly efficient means whereby we can visit every object on the heap, descending from root nodes down to leaves. Heap tagging allows us to associate objects with arbitrary 64-bit tags, useful for storing information about the status of each object (i.e., which test last read and wrote it) and each static field. Finally, heap access notifications allows us to register callbacks for the JVM to notify *ElectricTest* when specific objects or their primitive fields are written or read (when instance fields are accessed).

Observing New Heap Writes. For each test execution, *ElectricTest* needs to be able to efficiently determine what part of the heap was written by that test. We have optimized this process for cases where the majority of data created on the heap is not shared between tests (which we have found to be a common case). As objects are created on the heap during test execution, *ElectricTest* does nothing. At the end of each test, after performing a garbage collection, *ElectricTest* uses JVMTI to scan for all objects that have no tag associated with them (i.e., those not yet tagged by *ElectricTest*). Each untagged object is tagged with a counter indicating that it was created in the current test case. Objects are also tagged with the list of static fields from which they are accessible. Since the only objects that still exist after the test completes are those that can be shared between tests, this method avoids unnecessarily tagging and tracking objects that can't be part of dependencies.

Observing Heap Reads and Writes of Old Data. Aside from references on the stack, data on the JVM's heap

is accessed through fields of objects, static fields of classes, or array elements. The easiest type of heap access to observe is to the fields of objects, which *ElectricTest* accomplishes through JVMTI's field tracking system. For each class of object created in a previous test but still reachable in the current test, *ElectricTest* registers a callback through JVMTI to be notified whenever any fields of those objects are read or written.

When an object is read or written, *ElectricTest* checks its tag to see if it was last written or read in a previous test: if so, then it is marked as causing a dependency on the last test that wrote that object, and we note this dependence to report at the end of the test. This technique will detect both data dependencies (read after write) and anti-dependencies (write after read), reporting them independently.

Detecting reads and writes of static fields and array elements is more complicated, as there is no similar callback to use. Instead, *ElectricTest* relies on bytecode instrumentation, modifying the bytecode of every class that executes to directly notify *ElectricTest* of reads and writes. In its instrumentation phase, *ElectricTest* employs an intraprocedural data flow analysis to reduce the number of redundant calls that it makes to record reads and writes on the same value by inferring which arrays and fields have already been read or written before each instruction. *ElectricTest* also dynamically detects reads and writes through Java's reflection interface by intercepting all calls to the reflection API and adding a call to the *ElectricTest* runtime library to record the access.

Detecting Dependencies at Static Fields. At the end of each test, we perform a heap walk, rooted at every static field, visiting all objects that are reachable from each static field. We maintain a simple stop-list of common static fields within the Java API that are manually-verified as deterministically written and hence may be ignored in this process. These fields include fields such as `System.out`, which is the stream handler for standard output. While it is possible to modify these fields to point to a different object, their default value is always deterministically created. Therefore, a dependence on the default value of one of these fields can safely be ignored (a dependence on the *non-default* value is *not* ignored), since we can assume that this field would have the same value independent of which test first accessed it. This mechanism also allows developers to easily filter specific fields that are known to be data-dependent between executions, but benign (e.g. internals of logging mechanisms). A simple configuration file maintains a stop-list of fields for *ElectricTest* to ignore. *ElectricTest* marks all objects at the end of each test with the list of static fields that point to it.

3.2 Detecting External Dependencies

ElectricTest leverages the JVM's built-in *IOTrace* features to track access to external resources. When code attempts to access a file or socket, *ElectricTest* gets a callback identifying which file or network address is being accessed. All tests that access the same file or socket are marked as dependent. This relatively coarse approach is based on our observation that tests infrequently share access to the same file or network resource — or that if they do, they are dependent. While it would be possible to have a finer grained approach to detecting these dependencies (e.g. by tracing the exact data read and written), as our evaluation shows in the following section, the coarse grained approach is sufficient to allow for reasonable test suite acceleration in the projects we studied.

3.3 Reporting and Debugging Dependencies

Once all dependencies have been detected, *ElectricTest* can be used to help developers analyze and inspect them. While manual inspection is not required for sound test acceleration (the following section will describe how *ElectricTest* does this automatically), we imagine that in some cases developers will want to understand the dependencies between tests in their projects. For instance, perhaps some dependencies may be indicative of incorrectly written tests. We expect that in some cases developers may want to investigate dependencies to make sure that they are intentional. Alternatively, developers may want to mark some dependencies as benign: perhaps multiple tests intentionally share resources, but do so in a way that doesn't create a functional dependence. For instance, we have seen many test suites that intentionally share state between tests to reduce setup time: each test checks if the shared state is established and if not, initializes it, and resets it to this initial state when done.

ElectricTest supports developers analyzing dependencies by providing a complete stack-trace showing how a dependency occurred. Stack traces are trivially collected when a test reads data previously written since *ElectricTest* is detecting data dependencies in real-time, within the executing program and JVM. In this way, *ElectricTest* provides significantly more information than previous work in test dependency detection [40], which could only report that two tests were dependent.

3.4 Sound Test Acceleration

Given the list of dependencies between tests, we can soundly apply existing test acceleration techniques such as parallelization and prioritization. Naive approaches to both are straightforward, but may be sub-optimal. For instance, a naive approach for running a given test is to ensure that all tests that must run before it have just run (in order).

Haidry and Miller proposed several techniques for efficiently and effectively prioritizing test suites in light of dependencies [22]. Rather than consider prioritization metrics (e.g. line coverage) for a single test, entire dependency graphs are examined at once. Their techniques are agnostic to the dependency detection method (relying on manual specification in their work), and would be easily adapted to consume the dependencies output by *ElectricTest*.

We propose a simple technique to improve parallelization of dependent tests based on historical test timing information. Our optimistic greedy round-robin scheduler observes how long each test takes to execute and combines this data with the dependency tree to opportunistically achieve parallelism. Consider the simple case of ten tests, each of which take 10 minutes to run, all dependent on a single other test that takes only 30 seconds to run (but not dependent on each other). If we have 10 CPUs to utilize, we can safely utilize all resources by first running the single test that the others are dependent on on each CPU (causing it to be executed 10 times total), and then run one of the remaining 10 tests on each of the 10 CPUs. The testing infrastructure can then filter the unnecessary executions from reports.

ElectricTest generates schedules for parallel execution of tests using a greedy version of this algorithm, re-executing a single test multiple times on multiple CPUs when doing so would decrease wall time for execution. *ElectricTest* also

Table 3: Dependency detection times for DTDetector and *ElectricTest* using the same subjects evaluated in [40]. We show the baseline runtime of the test suite as well as the running time for three configurations of DTDetector: the 2-pair algorithm, the dependence-aware 2-pair algorithm, and the exhaustive algorithm. Execution times for DTDetector on Joda and with the Exhaustive algorithm (marked with *) are estimations based on the same methodology used by the authors of DTDetector [40].

	# of Tests		Testing Time (Seconds)					*ElectricTest* Speedup vs
				DTDetector				
Project	Classes	Methods	Baseline	All 2-pair	Dep-Aware Pairs	Exhaustive	*ElectricTest*	Dep-Aware
Joda	122	3875	16	*6,688,250	*657,144	*1E+308	2122	310X
XMLSecurity	19	108	22	28,958	5,500	*3E+174	57	96X
Crystal	11	75	4	3,050	874	*14E+108	22	40X
Synoptic	27	118	2	6,993	2,070	*2E+194	34	61X

Table 4: Dependencies detected by DTDetector (DTD) and *ElectricTest* (ET). For *ElectricTest*, we group dependencies, into tests that write a value which others read (W) and tests that read a value written by a previous test (R).

	Dependencies			ET Shared Resource Locations	
		ET			
Project	DTD	W	R	App Code	JRE Code
Joda	2	15	121	39	12
XMLSecurity	4	3	103	3	15
Crystal	18	15	39	4	19
Syntopic	1	10	117	3	14

can speculatively parallelize test methods by breaking simple dependencies. When one test depends on a simple (i.e., primitive) value from another test, *ElectricTest* will allow the dependent test to run separately from the test it depends on, simulating the dependent value. If *ElectricTest* runs the test writing that value and finds that it writes a different value than was replayed, the pair of tests are re-executed serially. In our evaluation that follows, we show that most dependencies are on a small number of tests, allowing this simple algorithm to greatly reduce the longest serial chain of tests to execute in parallel.

4. EVALUATION

We evaluated *ElectricTest* across three dimensions: accuracy, runtime performance, and impact on test acceleration techniques. For accuracy, we compare the dependencies detected by *ElectricTest* to the state-of-the-art tool, DTDetector [40]. In terms of performance, we measured the overhead of running *ElectricTest* on Java test suites compared to the normal running time of the test suite. Given that *ElectricTest* may report non-manifest dependencies (that is, those that need not be respected in order to maintain the integrity of the test suite), we are particularly interested in the impact of *ElectricTest*'s detected dependencies on test acceleration. To determine the impact of *ElectricTest* on test acceleration, we measured the longest chain of data dependencies and number of anti-dependencies in each of these large test suites to identify how effective test parallelization and selection could be when respecting the dependencies automatically detected by *ElectricTest*.

All of these experiments were performed in the same environment as our previous experiment in §2.3: Amazon EC2 r3.xlarge instances with 4 2.5Ghz CPUs and 30.5 GB of RAM (more details on this environment are in §2.3).

4.1 Accuracy

We evaluated the accuracy of *ElectricTest* by comparing the dependencies detected between test methods with those detected by Zhang et al.'s tool, DTDetector [40]. Table 4 shows the dependencies detected by each tool. *ElectricTest* detected all of the same dependencies identified by DTDetector, plus some additional dependencies. We therefore conclude that *ElectricTest*'s recall is at least as good as the existing tool, DTDetector.

We can directly attribute the additional dependencies to the different definition of dependencies employed by the two systems: *ElectricTest* detects all data dependencies, whereas DTDetector detects tests that have different outcomes when executed in a different order (manifest dependencies). Since not all data dependencies will result in manifest dependencies, we expect that *ElectricTest* reports more dependencies than DTDetector.

For the purposes of test acceleration, given the computational ability to execute DTDetector on the test suite under scrutiny, it may still be preferable to use it over *ElectricTest*. However, as discussed in §2.3, it is often infeasible to use DTDetector on projects of reasonable size. Moreover, in cases where developers want to debug a dependency, *ElectricTest* would still be preferable over DTDetector (which would only tell the developer that two tests had a dependency).

Interestingly, all dependencies detected by *ElectricTest* were caused by shared accesses to a very small number of resources (static fields in this case). Most of the data dependences were caused by references to static fields within the JRE made by JRE code (not by application code), and all such references were to fields of primitive types, allowing for the opportunistic parallelism described in §3.4. This is also interesting in that it may make manual investigation of dependencies by developers easier: even if many tests are dependent, the number of actual resources shared is small.

4.2 Overhead

We evaluated the overhead of *ElectricTest* on the same four subjects studied by Zhang et al. [40], in addition to the ten large Java projects described earlier in §2.3.

We reproduced Zhang et al.'s experiments [40] in our environment to provide a direct performance comparison between the two tools. Table 3 shows the runtime of the same four test suites evaluated by Zhang et al., presenting the baseline test execution time, the DTDetector execution time and the *ElectricTest* execution time. None of the DTDetector algorithms we studied provided reasonable performance on the *Joda* test suite, and the exhaustive technique was infeasible in all cases. Even in the case of the

Table 5: Dependencies found by *ElectricTest* on 10 large test suites. We show the number of tests in each suite, the time necessary to run the suite in dependency detection mode, the relative overhead of running the dependency detector (compared to running the test suite normally), the number of dependent tests, and the number of resources involved in dependencies. For dependencies, we report the number of tests that write a resource that is later read (W), the number of tests that read a previously written resource (W), and the longest serial chain of dependencies (C). For dependencies and resources in dependencies, we report our findings at the granularity of test classes and test methods.

	Number of Tests		Analysis Time	Analysis Relative	Test Dependencies						# Resources involved at level:	
					Classes			Methods				
Project	Classes	Methods	(Min)	Slowdown	W	R	C	W	R	C	Classes	Methods
camel	5,919	13,562	2,449	22.3X	1,977	3,465	1,356	4,790	8,399	1,695	4,944	5,490
crunch	62	243	165	9.4X	9	20	6	18	43	18	190	207
hazelcast	297	2,623	1,780	37.6X	174	200	186	1,163	1,261	1,482	941	1,020
jetty.project	554	5,603	184	9.2X	223	261	54	4,016	4,079	424	713	828
mongo-java-driver	58	576	103	1.4X	36	36	34	342	362	357	32	33
mule	2,047	10,476	9,698	82.6X	185	859	119	2,049	6,279	1,400	11,844	12,387
netty	289	4,601	338	5.4X	128	120	63	2,928	3,297	2,926	640	1,104
spring-data-mongodb	141	1,453	364	3.0X	114	130	110	1,407	1,404	1,401	1,469	1,489
tachyon	53	362	89	2.6X	9	13	9	55	93	13	125	157
titan	177	1,191	2,262	27.7X	118	126	46	429	877	40	1,433	1,562
Average	960	4,069	1,743	20.0X	297	523	198	1,720	2,609	976	2,233	2,428

dependence-aware optimized heuristics (which is not guaranteed to detect all dependencies), *ElectricTest* still ran significantly faster than DTDetector.

However, these test suites were all very small, with the longest taking only 22 seconds to run. We applied *ElectricTest* to the ten large open source projects with unisolated tests previously discussed in §2.3, recording the number of dependencies detected and the time needed to run the tool.

Table 5 shows the results of this study, showing the number of test classes and test methods in each project, along with the time needed to detect dependencies, the relative slowdown of dependency detection compared to normal test execution, and the number of dependent tests detected. For dependencies, we report dependence at both the level of test classes and test methods (in the case of test classes, we report the dependencies between entire test classes, and not the dependencies between the methods in the same test class). We report the number of tests writing values (W) that are later read by dependent tests (R), as well as the size of the longest serial chain (C). Finally, we report the distinct number of resources involved in dependencies between tests, both at the test class and test method level.

In general, far more tests caused dependencies (i.e., wrote a shared value) than were dependent (i.e., read a shared value). The longest critical path was fairly short (relative to the total number of tests) in almost all cases, indicating that test parallelization or selection may remain fairly effective. Given infinite CPUs to parallelize test execution across, the maximum speedup possible is restricted by this measure.

ElectricTest imposed on average a 20X slowdown compared to running the test suite normally to detect all dependencies between test methods or classes. In comparison, we calculated that on the same projects, DTDetector (using the pairwise testing heuristic) would impose on average a 2,276X slowdown when considering dependencies between test methods, or 279X between entire test classes (Table 3). *ElectricTest*'s overhead fluctuates with heap access patterns — in test suites that share large amounts of heap data between tests, *ElectricTest* is slower. The overhead also fluctuated somewhat with the average test method duration: since a complete garbage collection and heap walk had to occur after each test finishes, test suites consisting of a lot of very fast-executing tests (like in 'mule') had a greater slowdown.

We believe that *ElectricTest*'s overhead makes it feasible to use in practice, and note that it is still much less than DTDetector's, the previous system for detecting test dependencies [40].

4.3 Impact on Acceleration

Our approach may detect dependencies between tests that do not effect the outcome of tests. That is, two tests may have a data dependency, but this dependency may be completely benign to the control flow of the test.

Therefore, we take special care to evaluate the impact of dependencies detected by *ElectricTest* on test acceleration techniques, notably, test parallelization. In the extreme, if *ElectricTest* found data dependencies between every single test, techniques like test parallelization or test selection yield no benefits, since it would be impossible to change the order that tests ran in while preserving the dependencies.

We first evaluate the impact of *ElectricTest* on test acceleration techniques by examining the longest dependency chain detected in each project, shown under the heading 'C' in Table 5. In almost all projects, even if there were many dependent tests, the longest critical path was very short compared to the total number of tests. For example, while 4,079 of the 5,603 test methods in the jetty test suite depended on some value from a previous test, the longest dependency chain was 424 methods long. Across all of the projects, the average maximum dependency chain between test methods was 976 of an average 4,069 test methods and 198 between an average of 960 test classes. We find this result encouraging, as it indicates that test selection techniques can still operate with some sensitivity while preserving detected dependencies.

To quantify the impact of *ElectricTest*'s automatically detected dependencies on test parallelization we simulated the execution of each test suite running in parallel on a 32-core machine, distributing tests in a round-robin fashion in the

Table 6: Relative speedups from parallelizing each app's tests. Shown at the test class (C) and test method (M) while respecting *ElectricTest*-reported dependencies (with the naive scheduler and the greedy scheduler) in comparison to unsound parallelization without respecting dependencies.

	Naive ET		Greedy ET		Unsound	
Project	**C**	**M**	**C**	**M**	**C**	**M**
camel	4.6	6.5	6.9	9.8	16.1	18.0
crunch	4.8	3.0	7.8	3.1	12.4	15.5
hazelcast	1.2	2.1	1.2	2.6	12.9	20.0
jetty.project	6.1	6.9	6.1	6.9	9.5	17.0
mongo-java-driver	1.8	1.6	1.8	1.6	8.2	27.8
mule	9.6	7.9	9.6	13.8	17.0	18.0
netty	2.8	6.7	2.8	6.7	2.9	7.0
spring-data-mongodb	1.2	0.8	1.2	0.8	3.5	26.5
tachyon	3.9	4.3	3.9	15.5	5.3	26.9
titan	3.8	6.3	3.8	6.3	8.0	18.2
Average	4.0	5.0	5.0	7.0	10.0	19.0

same order they would typically run in. In this environment, there are 32 processes each running tests on the same machine, with each process persisting for the entire execution.

We simulated the parallelization of each test suite following three different configurations: without respecting dependencies ("unsound"), with a naive dependency-aware scheduler ("naive"), and the optimistic greedy scheduler described in §3.4 ("greedy"). The naive scheduler groups tests into chains to represent dependencies, such that each test is in exactly one group, and each group contains all dependencies for each test — this approach soundly respects dependencies but may not be optimal in execution time. Table 6 shows the results of this simulation, parallelizing at the granularities of test classes and test methods. We show the theoretical speedups for each schedule provided relative to the serial execution, where speedup is calculated as $T_{serial}/T_{parallel}$. Overall, the greedy optimistic scheduler outperformed the naive scheduler in some cases, and at times provided a speedup close to that of the unsound parallelization. In some cases, the dependency-preserving parallelization was faster when parallelizing at the coarser level of test classes. In these cases, there were so many dependencies at the test method level that the schedulers were generating incredibly inefficient schedules, requiring that some tests were re-executed many times. Also, this may have occurred in some cases because we assumed that the shortest amount of time that a single test could take was one millisecond: in the case of a single test class that took one millisecond that had several test methods, if we parallelized the test methods, we may assume a total time to execute longer than just running a test class at once.

We investigated the cases where *ElectricTest* didn't do as well, 'spring-data-mongodb' and 'mongo-java-driver' — both projects had very long dependency chains. Upon inspection, we found that most tests in each project *purposely* shared state between test cases for performance reasons. For instance, the mongo driver created a single connection to a database and reused that connection between tests to save on setup. The spring based project had a similar pattern.

These cases bring up an interesting point: sometimes tests may be intentionally data-dependent on each other. Especially in the case of short unit tests all testing the same large functional component, it is reasonable to expect that developers would intentionally re-use state to reduce the overall testing time. Thanks to its integration with the JVM, *ElectricTest* can easily be configured by developers to ignore particular dependencies at the level of static or instance fields.

4.4 Discussion and Limitations

There are several limitations to our approach and implementation. Because we detect dependencies of code running in the JVM, we may miss some dependencies that occur due to native code that is invoked by the test. While *ElectricTest* can detect Java field accesses from native code (through the use of field watches), it can not detect file accesses or array accesses from native code. However, none of the applications that we studied contained native libraries. It would be possible to expand our implementation to detect and record these accesses by performing load-time patching for calls to JNI functions for array accesses and system calls for file access to record the event.

When we detect external dependencies (i.e., files or network hosts), we assume that there is no collusion between externalities. For example, we assume that if one test communicates with network host A, and another test communicates with network host B, hosts A and B have no backchannel that may cause a dependency between the two tests. We have not built our tool to handle specialized hardware devices (other than those accessed via files or network sockets) that may be involved in dependencies. However, *ElectricTest* could easily be extended to handle such devices in the same manner as files and network hosts. Since *ElectricTest* is a dynamic tool, it will only detect dependencies that occur during the specific execution that it is used for: if tests exhibit different dependencies due to nondeterminism in the test, the dependency may not be detected. *ElectricTest* could be expanded to include a deterministic replay tool such as [9, 26] to ensure that dependencies don't vary.

There are also several threats to the validity of our experiments. We studied the ten longest building open source projects that we could find, in conjunction with four relatively short-building projects used by other researchers. These projects may not necessarily have the same characteristics as those projects found in industry. However, we believe that they are sufficiently diverse to show a cross-section of software, and show that *ElectricTest* works well with both long and short building software.

We simulated the speedups afforded by various parallel schedules of test suites. Due to resource limitations, we did not actually run the test suites in parallel. We assume that the running time of a test is constant, regardless of the order in which it is executed. Therefore we may expect that the speedup of the unsound parallelization is an overestimate: if multiple tests share the same state to save on time running setup code, then it may actually take longer to run the tests in parallel since the setup must run multiple times. However, we are confident that the various speedups predicted for dependency-preserving schedules are sound, as we do not believe that other external factors are likely to impact the running time of each test.

Similarly, we did not directly study the impact of the dependencies we detected on test selection or prioritization techniques, instead using the maximum dependency size as a proxy for selectivity. A more thorough study may have instead downloaded many different versions of each program

and performed test selection or prioritization on each version (based on results from the previous version) and then measured the impact of detected dependencies on these tools. Such a study also would show the practicality of caching *ElectricTest* results throughout the development cycle, so it need not be executed for each build. This caching might significantly reduce the performance burden of checking for test dependencies with each successive change to the program. Again, we were limited in resources to perform such a study, and believe that our use of maximum dependency size as an indicator for selectivity is sufficient.

5. RELATED WORK

Test dependencies are one root cause of the general problem of flaky tests, a term used to refer to tests whose outcome is non-deterministic with regards to the software under test [16, 29, 30]. Luo, et al. analyzed bug reports and fixes in 51 projects, studying the causes and fixes of 161 flaky tests, categorizing 19 (12%) of these to be caused by test dependencies [29]. *ElectricTest* could be used to automatically detect and avoid these dependency problems before they result in flaky tests.

ElectricTest is most similar to Zhang et al.'s DTDetector system, which detected *manifest dependencies* between tests by running the tests in various orderings [40]. A manifest dependency is indicated by a test having a different outcome when it is executed in a different order relative to the entire test suite. This approach required $O(n!)$ test executions for n tests, with best-case approximation scenarios at $O(n^2)$. *ElectricTest* instead detects data dependencies, where one test reads data that was last written by a previous test, and does not require running each test more than once, in a much more scalable approach.

Unlike *ElectricTest*, which observes and reports actual data dependencies between tests, Gyori et al.'s PolDet tool detects potential data sharing between tests by searching for data "pollution" — data left behind by a test that a later test may read (which may or may not ever occur) [21]. PolDet captures the JVM heap to an XML file using Java reflection and compares these XML files offline, while *ElectricTest* performs all analysis on live heaps, greatly simplifying detection of leaked data.

ElectricTest's technique for dependency detection is more related to work in Makefile (build) parallelization, such as EMake [33] or Metamorphisis [19]. These systems observe filesystem reads and writes for each step of the build process to detect dependencies between steps and infer which steps can be paralleled. In addition to filesystem accesses, *ElectricTest* monitors memory and network accesses.

While *ElectricTest* detects hidden dependencies between tests, there has also been work to efficiently isolate tests to ensure that dependencies do not occur. Popular Java testing platforms (e.g., JUnit [2] or TestNG [3] running with Ant [4] or Maven [5]) support optional test isolation by executing each test class in its own process, resulting in isolation at the expense of a high runtime overhead. Our previous work, VMVM, eliminates in-memory dependencies between tests without requiring running each test in its own process, greatly reducing the overhead for isolation [7]. While VMVM preserves the exact same semantics for isolation and initialization that would come by executing each test in its own process, other systems such as JCrasher [13] also isolate tests efficiently, although without reproducing the same exact semantics. If tests are already dependent on each other, but the goal is to isolate them, then *ElectricTest* could be used to identify which tests are currently dependent (and how), allowing a programmer to manually fix the tests so that they can run in isolation.

Other tools support test execution in the presence of test dependencies. However, all of these tools require developers to manually specify dependencies, a tedious and difficult process which is automated by *ElectricTest*. For instance, both the depunit [1] and TestNG [3] framework allow developers to specify dependencies between tests, while JUnit [2] allows developers to specify the order to run tests.

Test Suite Minimization identifies tests cases that may be redundant in terms of coverage metrics and removes them from the suite. Many heuristics and coverage metrics have been proposed to minimize test suites, although most approaches are limited by the strength of the coverage criteria used [11, 12, 23, 24, 27, 28, 37, 39]. Test selection approaches the same problem of having too many tests to run from a different angle by instead selecting only tests to run that have been impacted by changes in the application code since the last time that tests were executed [6, 10, 18, 20, 25]. Since test selection can be dangerous if additional tests are impacted by changes but not selected (i.e. due to imprecision in the coverage metrics used to determine impact), some may turn to test prioritization, where entire test suites are still executed, but tests most likely to be impacted by recent changes are executed first [14, 15, 35, 36, 38]. Haidry and Miller propose several test prioritization techniques that consider dependencies between tests when performing the minimization, but require developers to manually specify dependencies [22]. *ElectricTest* could be combined with each of these techniques to efficiently and automatically detect dependencies between tests, then safely accelerate them using test selection or prioritization.

6. CONCLUSIONS

While testing dominates long build times, accelerating testing is tricky, since test dependencies pose a threat to test acceleration tools. Test dependencies can be difficult to detect by hand, and prior to *ElectricTest*, there was no tool to practically detect them in all but the very smallest test suites (those which took less than several minutes to run normally). We have presented *ElectricTest*, a tool for detecting data dependencies between Java tests with an average slowdown of only 20X, where previous approaches would have been completely infeasible taking up to 10^{308} times longer to find all dependencies. We evaluated the accuracy of *ElectricTest*, finding it to have perfect recall compared to the previous approach in our study. Because not all data dependencies will influence the control flow of the data-dependent tests, we evaluated the impact of *ElectricTest* on test parallelization and selection, finding its dependency chains small enough to still allow for acceleration. The dependencies detected by *ElectricTest* can further be used by developers to gain insight into how their tests interact and fix unintentional dependencies.

7. ACKNOWLEDGEMENTS

Bell and Kaiser are members of The Programming Systems Laboratory, which is funded in part by NSF CCF-1302269, CCF-1161079, and NIH U54 CA121852. Bell was partially supported by Electric Cloud while completing this work.

8. REFERENCES

[1] Dependency and data driven unit testing framework for java. `https://code.google.com/p/depunit/`.

[2] Junit: A programmer-oriented testing framework for java. `http://junit.org/`.

[3] Next generation java testing. `http://testng.org/doc/index.html`.

[4] Apache Software Foundation. The apache ant project. `http://ant.apache.org/`.

[5] Apache Software Foundation. The apache maven project. `http://maven.apache.org/`.

[6] T. Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 134–142, New York, NY, USA, 1998. ACM.

[7] J. Bell and G. Kaiser. Unit Test Virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 550–561, New York, NY, USA, 2014. ACM.

[8] J. Bell, E. Melski, M. Dattatreya, and G. Kaiser. Vroom: Faster Build Processes for Java. In *IEEE Software Special Issue: Release Engineering*. IEEE Computer Society, March/April 2015. To Appear. Preprint: `http://jonbell.net/s2bel.pdf`.

[9] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press.

[10] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Inf. Softw. Technol.*, 51(1):16–30, Jan. 2009.

[11] T. Chen and M. Lau. A new heuristic for test suite reduction. *Information and Software Technology*, 40(5–6):347 – 354, 1998.

[12] T. Chen and M. Lau. A simulation study on some heuristics for test suite reduction. *Information and Software Technology*, 40(13):777 – 787, 1998.

[13] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[14] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 113–124, 2004.

[15] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[16] M. Fowler. Eradicating non-determinism in tests. `http://martinfowler.com/articles/nonDeterminism.html`, 2011.

[17] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov. Regression test selection for distributed software histories. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 293–309. Springer International Publishing, 2014.

[18] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 361–372, New York, NY, USA, 2014. ACM.

[19] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &; Applications*, OOPSLA '14, pages 599–616, New York, NY, USA, 2014. ACM.

[20] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[21] A. Gyori, A. Shi, F. Hairi, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 ACM International Symposium on Software Testing and Analysis*, 2015.

[22] S. Haidry and T. Miller. Using dependency structures for prioritization of functional test suites. *Software Engineering, IEEE Transactions on*, 39(2):258–275, Feb 2013.

[23] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 738–748, Piscataway, NJ, USA, 2012. IEEE Press.

[24] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[25] 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 *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

[26] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 385–386, New York, NY, USA, 2010. ACM.

[27] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Trans. Softw. Eng.*, 33(2):108–123, Feb. 2007.

[28] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–209, Mar. 2003.

[29] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM.

[30] A. M. Memon and M. B. Cohen. Automated testing of gui applications: Models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1479–1480, Piscataway, NJ, USA, 2013. IEEE Press.

[31] Oracle. Jvm tool interface. `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[32] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 241–251, New York, NY, USA, 2004. ACM.

[33] J. Ousterhout. 10–20x faster software builds. *USENIX ATC*, 2005.

[34] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–441, August 1996.

[35] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, pages 179–188, 1999.

[36] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.

[37] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 35–42, New York, NY, USA, 2005. ACM.

[38] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, Washington, DC, USA, 1997. IEEE Computer Society.

[39] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[40] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, M. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA '14, pages 384–396, New York, NY, USA, 2014. ACM.