File-Level vs. Module-Level Regression Test Selection for .NET

Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric
The University of Texas at Austin (USA), Microsoft (USA), Microsoft (USA), The University of Texas at Austin (USA)
vasic@utexas.edu,zuparvez@microsoft.com,almili@microsoft.com,gligoric@utexas.edu

ABSTRACT

Regression testing is used to check the correctness of evolving software. With the adoption of Agile development methodology, the number of tests and software revisions has dramatically increased, and hence has the cost of regression testing. Researchers proposed regression test selection (RTS) techniques that optimize regression testing by skipping tests that are not impacted by recent program changes. Ekstazi is one such state-of-the art technique; Ekstazi is implemented for the Java programming language and has been adopted by several companies and open-source projects.

We report on our experience implementing and evaluating Ekstazi#, an Ekstazi-like tool for .NET. We describe the key challenges of bringing the Ekstazi idea to the .NET platform. We evaluate Ekstazi# on 11 open-source projects, as well as an internal Microsoft project substantially larger than each of the open-source projects. Finally, we compare Ekstazi# to an incremental build system (also developed at Microsoft), which, out of the box, provides module-level dependency tracking and skipping tasks (including test execution) whenever dependencies of a task do not change between the current and the last successful build. Ekstazi# on average reduced regression testing time by 43.70% for the open-source projects and by 65.26% for the Microsoft project (the latter is in addition to the savings provided by incremental builds).

CCS CONCEPTS

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

KEYWORDS

Regression test selection, file-level dependencies, Ekstazi

ACM Reference format:
Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-Level vs. Module-Level Regression Test Selection for .NET. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 6 pages.
https://doi.org/10.1145/3106237.3117763

1 INTRODUCTION

Regression testing—running tests at a new project revision to ensure absence of regressions—is commonly used in industry to check the correctness of evolving software [16, 25, 29, 30]. The widespread adoption of Agile development methodology has led to a rapid increase in the number of tests and revisions, substantially increasing the cost of regression testing.

Regression Test Selection (RTS) techniques optimize regression testing by skipping tests that are not impacted by the changes introduced since the last test execution [33]. To this end, *test dependencies* are automatically tracked, i.e., discovered and remembered between executions. Dependency tracking can be done at different levels of granularity—statement, method, file, or module—leading to different trade-offs. Tests whose dependencies did not change since the last test execution may safely be skipped.

Researchers mostly studied RTS techniques with *fine-grained* dependencies, such as statements or methods [8, 9, 33]. These techniques collect dependencies either statically or dynamically. A technique that finds dependencies statically may be *unsafe* (i.e., it may miss to run some tests as it may not discover all true dependencies) [18]. On the other hand, a technique that collects dynamic fine-grained dependencies may introduce substantial *overhead* [13].

A common approach in industry is to use module-level RTS, i.e., track dependencies between project modules (arguably due to the high cost of fine-grained dependencies). Module-level RTS is currently available in many modern build systems [1, 2, 14], including Microsoft's CloudMake [5, 14]. Consider a project with two modules: `M1` and `M2`, where `M2` depends on `M1`. Assume also that there are two test modules `TM1` and `TM2` that exercise `M1` and `M2`, respectively. If a module changes (e.g., `M1`), CloudMake executes all tasks that transitively depend on the changed module; this includes its own test module (`TM1`), as well as the tests of all of its dependents (`TM2`). Module-level RTS is *imprecise*, i.e., runs more tests than necessary. This happens because dependencies are associated with a group of tests (i.e., a test module) rather than with individual tests, and dependencies are on modules rather than on *individual classes* inside those modules.

To balance the trade-offs between overhead, safety, and precision, we recently developed Ekstazi, an RTS tool for Java that collects dynamic file dependencies [13]. Specifically, Ekstazi collects file dependencies for individual test classes, and it computes checksums of each dependency. At a new revision, Ekstazi recomputes checksums, finds modified files, and runs test classes that depend on at least one of those files. Ekstazi has been shown effective in reducing both the number of executed test classes and testing time [13]. As a result, Ekstazi has been adopted by a number of open-source projects (e.g., Apache Camel and CXF [6]).

The current implementation of Ekstazi supports only languages that run on the Java Virtual Machine (JVM). Although the basic idea behind Ekstazi is fully generic and readily applies to other languages, in practice, however, implementing it for a new language may be hampered by various architectural differences between Java and that other language. In this paper, we address the challenges of

ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3117763

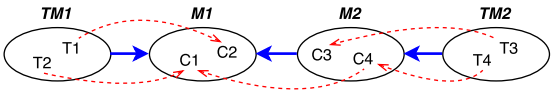

Figure 1: An example project used to illustrate file-level (dashed lines) and module-level dependencies (solid lines)

bringing the Ekstazi technique to .NET and provide a freely available implementation: Ekstazi# (https://github.com/marko-vasic/ekstaziSharp). Additionally, we compare the efficacy (in terms of end-to-end test execution time savings) of file-level RTS (as implemented in Ekstazi#) to module-level RTS that is typically provided by a build system capable of performing incremental builds.

We evaluated Ekstazi# on 11 open-source projects, totaling 313,166 lines of code and over 220 revisions. We measured (1) the number of non-executed (skipped) test methods, and (2) total savings in test execution time. On average (across all projects and all revisions), Ekstazi# executed only 32.62% of test methods (compared to *RetestAll*, i.e., running all tests at every revision), reducing total test execution time by 43.70%.

We also evaluated Ekstazi# at Microsoft, on an internal project substantially larger than each one of our open-source projects: it consisted of 45 program modules and 37 test modules (one module means one .dll file), totaling 2,730 .cs files and 391,992 lines of code. This project is built using Microsoft's latest incarnation of CloudMake, called *Concord* [24]. Concord is a general-purpose build system featuring dependency tracking and incremental builds (skipping tasks whose dependencies did not change), meaning that module-level RTS is provided out of the box. Our results show that *in addition* to the saving provided by Concord, Ekstazi# reduced the number of executed test methods by 75.61% and total test execution time by 65.26%.

2 FILE-LEVEL VS. MODULE-LEVEL RTS

This section illustrates file-level and module-level RTS techniques and describes relevant features of Concord and Ekstazi.

Figure 1 shows an example project that includes two modules (`M1` and `M2`) and two test modules (`TM1` and `TM2`).

Concord is a general-purpose (language-agnostic) build system. That means it only knows about files, build tasks, dependencies between tasks, and how to execute each build task. To build the example in Figure 1, the user has to specify that the program modules (`M1`, `M2`) are built by invoking the C# compiler, and the test modules (`TM1`, `TM2`) are built by first invoking the C# compiler and then running XUnit on the compiled DLL. Task dependencies (depicted with solid lines in Figure 1) must also be statically specified by the user (which is common for build systems supporting deterministic and cacheable builds, including Microsoft's CloudMake [5, 14], Google's Bazel [1], and Facebook's Buck [2]).

When a build is requested, Concord detects modified files and computes a transitive closure of affected modules; whichever test modules are affected are executed. As a result, if `M1` changed in any way possible (e.g., a new, currently unused, class is added), all test classes from modules `TM1` and `TM2` are executed. This is not due to a deficiency of Concord; rather, this is a fundamental property of a language-agnostic build system, which, as such, can only offer module-level RTS capabilities.

Ekstazi is a dynamic RTS technique which, in contrast, collects dependencies at a much finer level of granularity: for each test class, Ekstazi dynamically observes and collects files that get used (in any shape or form) during that test execution. As a result, file-level dependencies are collected (shown with dashed lines in Figure 1). Additionally, instead of detecting changes at the file level, which is the best a language-agnostic build system can do, Ekstazi computes *smart checksums* (excluding trivia like debug information) for each class. When a build is requested, smart checksums are used to detect changed classes, and previously collected dependencies are used to detect affected tests that must be executed.

Going back to our running example, if only class `C2` has changed, Ekstazi executes only `T1`, while Concord executes all tests in modules `TM1` and `TM2` (i.e., `T1`, `T2`, `T3`, `T4`).

3 EKSTAZI#

Ekstazi# is an implementation, written in C#, of the Ekstazi technique for .NET.

Instrumentation: Before executing tests, all classes are instrumented to enable dependency collection. Ekstazi# instruments constructors and accesses to static methods and fields, using the *Mono Cecil* library, which is similar to ASM for Java. Unlike Ekstazi which instruments code dynamically (when a class is loaded), Ekstazi# instruments code statically (before executing tests), because dynamic class rewriting is not supported in .NET.

Testing framework: Frequently, test classes for a single module are run inside a single Virtual Machine. Therefore, to be able to capture dependencies for each test class individually, we have to detect the beginning and end of each test class. We do this by instrumenting test class in a way that is specific to a testing framework (we currently support NUnit and XUnit). In Ekstazi#, we also support *selection granularity* on test methods, i.e., capturing dependencies for individual test methods. We evaluate both (test class and method) selection granularities.

Checksum: Ekstazi# implements *smart checksums*, which ignore debug info for each class. Optionally, methods in a class can be sorted by name, which adds resilience to simple formatting changes; in our experiments, however, this provided no benefits. To compute checksums, Ekstazi# relies on a 3rd party open-source library (https://crc32c.angeloflogic.com) and its implementation of CRC-32C.

4 EVALUATION

This section describes our evaluation of Ekstazi#. To this end, we answer the following research questions using open-source projects:

- **RQ1:** What are the average savings in test execution time?
- **RQ2:** What are the average savings in number of executed tests?
- **RQ3:** What difference does the smart checksum make?
- **RQ4:** How does selection granularity affect performance?

Additionally, we evaluated Ekstazi# on a large project at Microsoft by answering the following questions:

- **RQ5:** What are the average savings in test execution time and number of tests for a large industrial project?
- **RQ6:** How much savings can Ekstazi# provide on top of the module-level savings of Concord?

Execution Platforms: We performed the experiments with the open-source projects on a VM with 1-core, 2GB main memory, running Microsoft Windows 10. The host machine is a 4-core Intel(R)

Table 1: Metrics of the Projects Used in the Evaluation

Project	URL[https://github.com/]	SHA	LOC	#Files	Tests	
					classes	methods
Abot	sjdirect/abot.git	c07b71af89	10,737	108	39.20	421.20
DynamicExpresso	davideicardi/DynamicExpresso.git	186219674d	5,819	63	24.05	205.20
FCLP	fclp/fluent-command-line-parser.git	d8eade0f36	6,832	149	6.00	137.70
FluentValidation	JeremySkinner/FluentValidation.git	445f6390f3	15,667	226	63.00	531.95
JsonFx	jsonfx/jsonfx.git	1c72c653a2	34,839	151	35.10	781.10
MoreLinq	morelinq/MoreLINQ.git	d987f0ef30	9,784	161	64.50	494.75
Nancy	NancyFx/Nancy.git	ef567c2930	69,120	958	135.45	1,600.20
NewtonsoftJson	JamesNK/Newtonsoft.Json.git	218c25c859	116,373	628	183.00	2,253.25
OptiKey	OptiKey/OptiKey.git	84ea4049a4	23,440	292	50.00	190.00
RequestReduce	mwrock/RequestReduce.git	ddd623b5ee	16,477	137	13.00	97.00
Stateless	dotnet-state-machine/stateless.git	7c03035f10	4,078	56	12.00	111.95
Avg.	N/A	N/A	28,469.63	266.27	56.84	620.39
Σ	N/A	N/A	313,166	2,929	625.30	6,824.30

Core(TM) i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 14.04 LTS. For the experiments at Microsoft, we used a machine with 12-core Intel(R) Xeon(R) E5-1650 v3 CPU @ 3.50GHz with 32GB of RAM, running Microsoft Windows 10.

4.1 Projects

4.1.1 Open-source Projects. To the best of our knowledge, this is the first extensive study of regression testing on C# projects. Therefore, one contribution (although minor) of this paper is a collection of C# projects that could be a valuable starting point for other researchers working on program analysis for .NET languages.

Table 1 shows some basic information about the projects used in our study, including name, git URL, SHA of the latest used revision in our experiment, number of lines of code (LOC) computed using the `cloc` tool, number of all source (.cs) files, and the average number of test classes and test methods per revision; the last two columns show the average values computed across 20 revisions that are used in the experiments. The last two rows show (where applicable) the average and total values across all projects.

The C# projects used in this evaluation vary in size (from 4,078 to 116,373 LOC), number of test classes (from 6 to 183 test classes), and domain of application.

4.1.2 Microsoft's Project. We deployed Ekstazi# at Microsoft and evaluated its benefits on a large project (referred to as *MSProj*) with 45 program modules and 37 test modules, totaling 391,992 LOC, across 2,730 .cs files.

4.2 Experiment Setup

For each project, we chose the latest revision available, and performed the following steps (similar to recent work on RTS [13]):

a) Select 20 (40 for MSProj) buildable revisions in the software history that are closest to the latest available revision.
b) Restore (i.e., checkout) the oldest revision among the revisions chosen in the previous step.
c) Execute tests with the RetestAll technique; this step reports the total number of available tests and measures execution time.
d) Integrate Ekstazi#, select affected test classes, instrument classes, and execute selected tests. This step collects new dependencies for each executed test class, and reports the number of selected test methods (#Sel), the overhead times spent selecting affected tests (A[s]), instrumenting code (I[s]), and executing tests and collecting new dependencies ((E+C)[s]).
e) If this is not the latest revision, move one revision forward, and start again from step c).

4.3 Answers to the Research Questions

4.3.1 RQ1: What are the Average Savings in Test Execution Time. The key metric for evaluating the benefits of an RTS technique is the savings in end-to-end execution time [13]; in our experiments, we do not include compilation in the end-to-end time. Table 2 shows the execution time for Ekstazi# and RetestAll. Columns 4, 5, and 6 show analysis time, instrumentation time, and execution+collection time, respectively. Each column shows the *cumulative time* across 20 revisions. Column 7 shows cumulative end-to-end time for Ekstazi#. Column 9 shows cumulative end-to-end time for RetestAll. Finally, the last column show the percentage of end-to-end execution time (across all revisions) of Ekstazi# to RetestAll (time[%] = Total[s] / E[s] * 100).

Our results show that Ekstazi# is effective, i.e., reduces the overall time compared to RetestAll, for *all* projects. For several cases, namely JsonFx and OptiKey, the savings are not substantial. These projects have high ratio of LOC over the number of test classes and/or short test execution times (Table 1), so the overhead of instrumentation and analysis becomes quite high, and Ekstazi# can barely make up for it during test execution; this is not surprising. In sum, across all projects, Ekstazi# reduces test execution time to 56.30% on average.

4.3.2 RQ2: What are the Average Savings in Number of Executed Tests? Table 2 also shows the total number of executed tests with Ekstazi# and RetestAll in columns 3 and 8, respectively. In addition, in column 10, we show *test selection ratio*, i.e., a number of selected tests over the total number of executed tests with RetestAll (m[%] = #Sel(Ekstazi#) / #Sel(RetestAll) * 100).

Test selection ratio (in terms of executed test methods) ranges from 8.55% to 53.39%, and that it is on average 32.62%.

Table 2: Comparison of Ekstazi# (Class Selection with Smart Checksum) and RetestAll Approaches

Project	#Revs	Ekstazi#					RetestAll		Test Selection	
		#Sel	*A*[s]	*I*[s]	(*E*+*C*)[s]	Total[s]	#Sel	*E*[s]	m[%]	time[%]
Abot	20	1,455	15.97	12.94	447.71	476.62	8,424	1,525.15	17.27	31.25
DynamicExpresso	20	1,267	1.54	1.22	46.85	49.62	4,104	128.11	30.87	38.73
FCLP	20	1,204	4.26	4.52	15.09	23.89	2,754	30.99	43.71	77.08
FluentValidation	20	3,013	5.50	3.01	26.88	35.39	10,639	84.43	28.32	41.91
JsonFx	20	4,648	4.51	5.19	9.61	19.32	15,622	19.50	29.75	99.07
MoreLinq	20	3,922	8.64	4.88	302.87	316.40	9,895	786.27	39.63	40.24
Nancy	20	8,563	33.19	14.63	516.98	564.81	32,004	1,337.41	26.75	42.23
NewtonsoftJson	20	24,061	45.44	25.35	255.30	326.10	45,065	433.76	53.39	75.17
OptiKey	20	1,348	30.08	16.54	34.18	80.81	3,800	84.79	35.47	95.30
RequestReduce	20	166	8.72	6.30	114.68	129.70	1,940	468.09	8.55	27.70
Stateless	20	1,012	2.34	2.27	44.00	48.62	2,239	96.02	45.19	50.63
Avg.	20	4,605.36	14.56	8.80	164.92	188.29	12,407.81	454.04	32.62	56.30
Σ	220	50,659	160.19	96.85	1,814.15	2,071.28	136,486	4,994.52	N/A	N/A

4.3.3 RQ3: What Difference Does the Smart Checksum Make? Table 3 shows the results of Ekstazi# on the open-source projects, when we turn *off* the smart checksum (Section 3). We show test selection ratio (Column 2) and percent of time to run tests compared to RetestAll (Column 3). These values should be compared with the values in the last two columns in Table 2.

Table 3: Class Selection with Regular Checksum

Project	m[%]	time[%]
Abot	17.27	30.23
DynamicExpresso	35.84	43.75
FCLP	43.71	81.25
FluentValidation	28.33	44.84
JsonFx	33.20	102.10
MoreLinq	39.63	35.67
Nancy	27.50	41.64
NewtonsoftJson	53.68	70.10
OptiKey	35.47	114.68
RequestReduce	8.55	26.94
Stateless	45.19	48.76
Avg.	33.48	58.17

Our results show that, on average, slightly fewer tests are run when using smart checksum, 32.62% compared to 33.48%, saving 0.86%. Analysis time is slightly higher with the smart checksum, due to additional complexity of checksum computation. However, execution time is lower due to savings in number of tests run. In total, the run with smart checksum takes 56.30% to execute, while the run with regular checksum takes 58.17%, saving 1.87%.

4.3.4 RQ4: How does Selection Granularity Affect Performance. Table 4 shows the results of Ekstazi# if we configure the tool to use method-level, rather than class-level, selection granularity (Section 3). As before, we show the test selection ratio (Column 2) and percent of time to run tests with Ekstazi# compared to RetestAll (Column 3). These values should be compared with the values in the last two columns in Table 2.

Our results show that, on average, 24.79% of test methods are run in case of method-level selection granularity, as opposed to 32.62% when class-level selection granularity is used, saving 7.83% of executed methods. Total execution time, however, is higher—76.05% (of RetestAll) vs. 56.30%—due to increased overhead associated with method-level selection. This is not surprising and it matches our prior findings for Ekstazi [13]). The default for Ekstazi#, hence, is to use class-level selection granularity.

4.3.5 RQ5: What are the Average Savings in Test Execution Time and Number of Tests for a Large Industrial Project. Table 5 shows test execution time for MSProj (described in Section 4.1.2) with Ekstazi# over 40 revisions. In all our experiments, we run tests sequentially to provide stable measurement.

Table 4: Method Selection with Smart Checksum

Project	m[%]	time[%]
Abot	9.25	26.99
DynamicExpresso	30.67	40.56
FCLP	43.17	91.86
FluentValidation	23.42	108.53
JsonFx	25.69	172.46
MoreLinq	26.48	38.48
Nancy	12.79	37.82
NewtonsoftJson	45.63	112.46
OptiKey	5.68	148.74
RequestReduce	7.21	11.31
Stateless	42.78	47.36
Avg.	24.79	76.05

Table 6 shows the comparison between RetestAll and Ekstazi# on MSProj; we discuss the second row in the next section. RetestAll in this case simply uses XUnit to execute *all* available tests in every revision. Column 3 shows the number of selected tests (i.e., all tests in case of RetestAll). Column 4 shows time to execute selected tests (i.e., all tests in case of RetestAll). Based on columns 3 and 4, we can see that Ekstazi# (compared to RetestAll) reduces the number of executed test methods by 87.52% and the execution time by 91.05%. (Note that these two numbers are not shown in any table).

4.3.6 RQ6: How Much Savings can Ekstazi# Provide on Top of the Module-Level Savings of Concord. Concord supports incremental and cached builds, so the RetestAll strategy is almost never used in practice at Microsoft. Hence, here we evaluate whether Ekstazi# can provide savings over what already comes for free from Concord (which basically amounts to module-level RTS).

The setup for this experiment is similar to the setup we used before, with a few important differences. Before building the project for the first time (i.e., when the oldest revision is selected), we clear

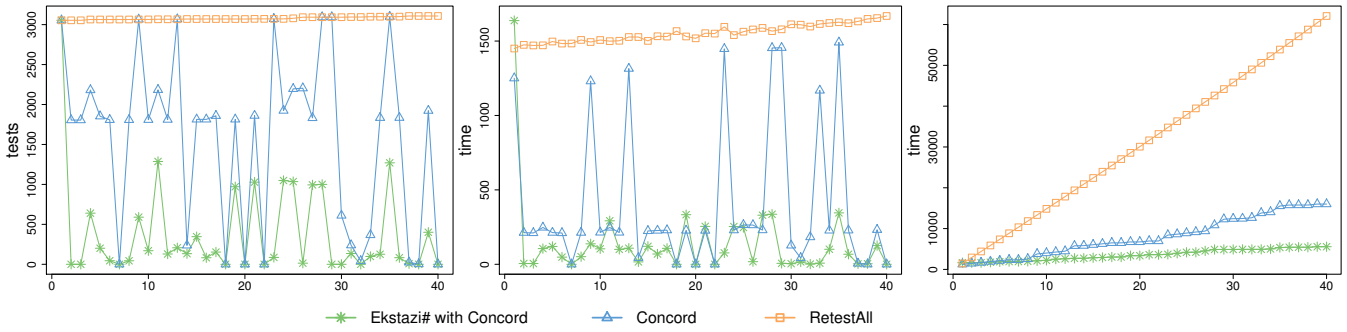

Figure 2: Selected tests (left), execution time (center), and cumulative time (right) for MSProj

Table 5: Breakdown of Time for MSProj

	A[s]	*I*[s]	(*E*+*C*)[s]	Total[s]
Ekstazi#	179.78	267.43	5,112.92	5,560.14

Table 6: Results for MSProj; Last two columns (m[%] and time[%]) for Ekstazi# are Computed Relative to Concord

	#Revs	#Sel	time[s]	m[%]	time[%]
RetestAll	40	123,154	62,090.62	N/A	N/A
Concord	40	63,039	16,003.72	51.18	25.77
Ekstazi#	40	15,379	5,560.14	24.39	34.74

Concord's cache, so that the experiment starts with a clean state, and the first build executes everything. When switching between project revisions, we keep the cache intact. Concord uses the cache to fetch the results of build tasks whose dependencies didn't change; in practice, that means not executing tests whose module-level dependencies were not modified. After a project revision is built, we use Concord's analysis tools to determine exactly which test modules were executed. We pass those test modules to Ekstazi# and let it instrument them and execute only the affected test classes.

Table 6 shows results for Concord (second row). Column 5 shows reduction in the number of executed tests methods (Concord compared to RetestAll and Ekstazi# compared to Concord). Ekstazi# reduces the number of test methods by 75.61% and the execution time by 65.26% compared to Concord. This result shows that file-level RTS can be a great addition to modern build systems that support module-level RTS.

There are a number of reasons why Ekstazi# selects quite fewer tests. For example, our project contains a utility module (assembly) consisting of a diverse set of functionalities used across the whole project (which is a common pattern in software development in general). For an assembly to make any use of those utilities (even a single function), that assembly has to declare a dependency on the whole utilities module. Consequently, whenever the utilities change, e.g., a new function is added (which tends to happen often), Concord executes all tests of all dependent modules. Ekstazi#, in contrast, automatically collects more detailed dependencies (between test classes and program classes), so it can be significantly more precise when it comes to selecting tests that are sufficient for ensuring absence of any regressions. Comparison among RetestAll, Concord, and Ekstazi# is visualized in Figure 2. On the x-axis we have the revision number (from older to newer). On the y-axis we have the number of executed test methods (Figure 2 - left); end-to-end time for a revision (Figure 2 - center); and cumulative end-to-end time over revisions (Figure 2 - right).

5 THREATS TO VALIDITY

External: Our results may not generalize to projects not used in this study. To mitigate this threat, we extensively searched various repository hosting services for open-source C# projects. These projects varied in size, number of tests, testing framework, and domain of application. Additionally, we evaluated Ekstazi# on a large Microsoft's project.

We ran experiments for each group of projects (open-source vs. Microsoft) on a single machine. The results could show different savings on another platform. To mitigate this threat, we run a subset of projects on different hardware and observed similar savings.

Internal: Our implementation of Ekstazi# may contain bugs. We have written a number of unit tests for Ekstazi#, which are similar to those of Ekstazi, to increase our confidence in the correctness of the implementation and its safety.

Construct: We have used 20 revisions per project (40 for MSProj); the results could be different if we used longer sequences of revisions or if we used a different window of revisions. Our experience has shown that extending the sequences of revisions was not feasible for most of the projects due to missing dependencies for older project versions [31] or due to changes in testing frameworks.

Although many RTS techniques have been proposed [8, 9, 33], we compare only class and method selection granularity. To the best of our knowledge, no other RTS technique has been implemented in a publicly available tool for C#.

6 RELATED WORK

RTS techniques: Most closely related tool is Ekstazi [12, 13] that we extensively discussed throughout the paper. Other related tools are those that dynamically collect test dependencies on statements, methods [4, 34], or files (but require that each test is run in a separate VM) [3]. Several tools collect test dependencies statically. For example, Ren et al. [21] developed Chianti, a tool that statically analyzes source code and collects dependencies on methods. Kung et al. [17] proposed static RTS based on the class firewall, i.e., statically computed set of classes that may be affected by changes. Recently, other researchers have evaluated class firewall approach for Java [18] and C# [19]. Orso et al. [20] combined class firewall with method dependencies. Skoglund and Runeson evaluated the

class firewall approach [26] and combined it with dynamic class dependencies. Later they showed that computing the class firewall is not necessary if dependencies on classes are dynamically collected [27]. We are the first to implement Ekstazi for .NET.

RTS in industry: Zheng et al. [35] proposed a black-box RTS technique to identify changes in API of the given components and evaluated the technique on one component at ABB. Elbaum et al. [7] demonstrated benefits of (static) RTS on several Google projects. Herzig et al. [16] introduced THEO, an RTS technique based on a cost model. Our goal was to implement an existing RTS technique for .NET and report our experience. Furthermore, we have studied a dynamic RTS technique. Based on prior work, Ekstazi is safer and more precise than static approaches [18] and, as shown in this paper, more precise than techniques that analyze entire modules.

Other work on regression testing: Many other techniques have been proposed to improve regression testing. Test-case prioritization techniques reorder tests to identify failing tests faster [23, 33]. For example, Srivastava and Thiagarajan [28] introduced Echelon at Microsoft; Echelon orders tests based on the basic block coverage (i.e., tests that cover more basic blocks are run first). Test suite reduction techniques identify redundant tests that should be removed from the test suite [22, 33]. While our work focuses on RTS, our infrastructure could be a base for test-case prioritization and test suite reduction techniques that are based on coverage.

Build systems: Many existing build systems: Make [15], CloudMake [5], CloudBuild [11], and Bazel [1] compute dependencies statically. This leads to overapproximation, i.e. a build target may not depend on all specified dependencies. In such cases the build target will be triggered even if an unrelated file gets modified; these build targets are considered underutilized. Vakilian et al. [32] aim to address underutilization by searching for an optimal repartitioning of build targets. Furthermore, to improve the build utilization Erdweg et al. [10] proposed Pluto, a build system with dynamic dependencies. Ekstazi# is more precise then build systems that statically compute dependencies, and those that do it dynamically have not yet made breakthrough to the commercial usage, even if they did our tool would still be more effective since it collects dependencies on a level of classes or methods that are part of a bigger file (DLL or EXE) that encompasses many classes and methods.

7 CONCLUSION

We presented Ekstazi#, a tool for regression test selection that implements the Ekstazi technique (originally for JVM) for .NET. We discussed challenges faced due to the underlying differences between JVM and .NET. We evaluated Ekstazi# on 11 open-source projects and one large project at Microsoft. Our results showed that Ekstazi# can substantially reduce the test execution time, even when compared to a module-level RTS technique, which is supported by modern build systems. In conclusion, Ekstazi# is a robust tool that would be a valuable addition to any modern build system and any project with long test execution.

Acknowledgments. We thank the fellow students of EE 382C at The University of Texas at Austin for constructive discussions on the material presented in this paper. We also thank Ahmet Celik, Nima Dini, and Sarfraz Khurshid for their feedback on this work. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363 and CCF-1652517.

REFERENCES

[1] Bazel build system. https://bazel.build.
[2] Buck home page. https://github.com/facebook/buck.
[3] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *FSE*. TO APPEAR.
[4] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: A System for Selective Regression Testing. In *ICSE*. 211–220.
[5] Maria Christakis, K. Rustan Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language. In *FM*. 643–657.
[6] Ekstazi home page. http://www.ekstazi.org.
[7] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE*. 235–245.
[8] Emelie Engström and Per Runeson. 2010. A Qualitative Survey of Regression Testing Practices. In *PROFES*. 3–16.
[9] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *IST* 52, 1 (2010), 14–30.
[10] Sebastian Erdweg, Moritz Lichter, and Weiel Manuel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *OOPSLA*. 89–106.
[11] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *ICSE*. 11–20.
[12] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *ICSE, Demo*. 713–716.
[13] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *ISSTA*. 211–222.
[14] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014. Automated Migration of Build Scripts Using Dynamic Analysis and Search-based Refactoring. In *OOPSLA*. 599–616.
[15] GNU Make home page. https://www.gnu.org/software/make/.
[16] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less without Sacrificing Quality. In *ICSE*. 483–493.
[17] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. *JOOP* 8, 2 (1995), 51–65.
[18] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*. 583–594.
[19] Nashat Mansour and Wael Statieh. 2009. Regression Test Selection for C# Programs. *Adv. Soft. Eng.* 2009 (2009), 1:1–1:16.
[20] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *FSE*. 241–251.
[21] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA*. 432–448.
[22] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical Studies of Test-Suite Reduction. *STVR* 12, 4 (2002), 219–249.
[23] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*. 179–188.
[24] Wolfram Schulte. 2016. Changing Microsoft's build: Revolution or evolution. In *ASE, Abstract*. 2–2.
[25] August Shi, Suresh Thummalapenta, Shuvendu Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *ICSE*. 689–699.
[26] Mats Skoglund and Per Runeson. 2005. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *ISESE*. 74–83.
[27] Mats Skoglund and Per Runeson. 2007. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. *IJSEKE* 17, 3 (2007), 359–378.
[28] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *ISSTA*. 97–106.
[29] Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.
[30] Tools for Continuous Integration at Google Scale. http://www.youtube.com/watch?v=b52aXZ2yi08.
[31] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and Back Again: Can you Compile that Snapshot? *JSEP* (2017), to appear.
[32] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *ICSE*. 123–133.
[33] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012), 67–120.
[34] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. 23–32.
[35] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley. 2005. An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code Is Not Available. In *ISSRE*. 225–234.