Resurgence of Regression Test Selection for C++

Ben Fu[1], Sasa Misailovic[2], and Milos Gligoric[1]
[1]The University of Texas at Austin, [2]University of Illinois at Urbana-Champaign
Email: ben_fu1@utexas.edu, misailo@illinois.edu, gligoric@utexas.edu

***Abstract*—Regression testing – running available tests after each project change – is widely practiced in industry. Despite its widespread use and importance, regression testing is a costly activity. Regression test selection (RTS) optimizes regression testing by selecting only tests affected by project changes. RTS has been extensively studied and several tools have been deployed in large projects. However, work on RTS over the last decade has mostly focused on languages with abstract computing machines (e.g., JVM). Meanwhile development practices (e.g., frequency of commits, testing frameworks, compilers) in C++ projects have dramatically changed and the way we should design and implement RTS tools and the benefits of those tools is unknown.**

We present a design and implementation of an RTS technique, dubbed RTS++, that targets projects written in C++, which compile to LLVM IR and use the Google Test testing framework. RTS++ uses static analysis of a function call graph to select tests. RTS++ integrates with many existing build systems, including AutoMake, CMake, and Make. We evaluated RTS++ on 11 large open-source projects, totaling 3,811,916 lines of code. To the best of our knowledge, this is the largest evaluation of an RTS technique for C++. We measured the benefits of RTS++ compared to running all available tests (i.e., retest-all). Our results show that RTS++ reduces the number of executed tests and end-to-end testing time by 88% and 61% on average.

***Index Terms*—Regression test selection, static analysis, call graph, LLVM, Google Test**

I. Introduction

Regression testing – running available tests at each project revision to check the correctness of recent project changes – is widely practiced in industry. The widespread availability of continuous integration systems simplified build configurations to enable regression testing, and continuous integration services, e.g., TravisCI [45], provide necessary resources even to open-source projects. Although regression testing is important, it is a rather costly activity, and this cost tends to increase with the increase in the number of tests and the frequency of project changes [43], [44]. Many large software organizations, including Apache, Facebook, Google, Microsoft, Salesforce, and Uber have reported high cost of regression testing and have been adopting various techniques to reduce this cost [7], [13], [15], [21], [23], [42].

Regression test selection (RTS) optimizes the regression testing activity by *selecting* tests that are *affected* by recent project changes and skipping to run the remaining subset of tests [5], [14], [39], [50]. Traditionally, RTS techniques keep a mapping from each test to all code elements (e.g., statements, basic blocks, functions, classes) that the test might use and select those tests (at a new project revision) that depend on any modified code element. The mapping from each test to code elements can be obtained either *statically* (without running the test) or *dynamically* (during test execution on the old project revision). The code elements on which dependencies are kept determines the *granularity* of an RTS technique, e.g., statement-level, function-level.

RTS techniques have been studied for over four decades, and several surveys summarize RTS status and progress [5], [14], [50]. Researchers and practitioners have studied RTS techniques for various programming languages, including C/C++, C#, and Java (e.g., [10], [16], [27], [34], [35], [47], [52]). These techniques kept dependencies on various code elements, such as basic blocks, functions/methods, and classes (e.g., [10], [16], [42]), and used both static and dynamic analysis (e.g., [16], [27], [52]).

Motivation. Most of the initial work on RTS techniques was focused on languages that compile to an executable file, e.g., C/C++, but work in the last decade has mostly focused on RTS for languages with abstract computing machines, such as Java and C#. Although C++ is still a widely used programming language and projects written in C++ come with many tests, researchers arguably focused on Java and C# as they were easier to analyze and come with an abundance of libraries. Despite the revolution of C++ compilers (e.g., the popularity of LLVM), testing frameworks (e.g., the widespread use of Google Test), and development processes (the popularity of GitHub and continuous integration services), the impact of these factors on RTS *design, implementation, and provided benefits* is unknown. Additionally, recent practices to evaluate RTS implementations based on *end-to-end* time (i.e., time to run the RTS tool and selected tests) were not used for RTS tools that target C++ projects for decades.

Technique. We designed, implemented, and evaluated the first RTS technique, named RTS++, which supports projects that compile to LLVM IR (intermediate representation) [28], use Google Test, and follow modern development practices. RTS++ is the first RTS technique for C++ based on *static function-level* call graph analysis. At a new project revision, RTS++ analyzes code changes and runs those tests that depend on one of the modified/deleted/added functions. RTS++ ensures to select an appropriate set of tests even in the presence of dynamic dispatch; this is achieved by a separate analysis of call graphs obtained for both the old and new project revisions. RTS++ also supports changes in macros and templates by design, because it analyzes LLVM IR. RTS++ is influenced by prior RTS techniques that analyze class-firewall, control-flow graphs, and dangerous-edges [24], [34], [35], [39]. The key differences include the level on which RTS++ keeps dependencies, the way analysis is performed, and support for LLVM IR and Google Test.

We implemented RTS++ as an LLVM compiler pass [29]. The benefit of this approach is that RTS++ can be extended to support other languages that compile to LLVM bitcode. RTS++ blends nicely with Google Test and supports all test types. However, most of RTS++ is independent of Google Test and could be integrated with other testing frameworks, such as the Boost Test Library [6]. RTS++ currently supports AutoMake, CMake, and Make build systems.

Evaluation. We performed an extensive evaluation of RTS++ on 11 open-source projects available on GitHub, totaling 3,811,916 lines of code and 1,709 test cases. To the best of our knowledge, this is the largest evaluation of an RTS technique for C++. This is also the first evaluation of an RTS for C++ that uses each revision rather than project releases. We use (up to) 50 latest revisions of each project.

To assess the benefits of RTS++, we measured savings compared to *retest-all* strategy – running all test at each project revision (i.e., a strategy oblivious of the program changes). We compared the number of executed tests and end-to-end testing time. Our results show that RTS++ can provide substantial savings. RTS++ reduced the number of tests of up to 97.20% (88% on average) compared to retest-all. RTS++ reduced the end-to-end testing time up to 88.09% (61% on average) compared to retest-all (considering only those projects when testing time is substantial). These were surprising findings as recent work for Java showed that using static method call graph, which is analogous to function members in C++, provides no benefits [16], [27], [51].

The main contributions of this paper include:

⋆ RTS++, an RTS technique, based on a static function-level call graph analysis.

⋆ Implementation of RTS++ for projects that compile to LLVM IR and use the Google Test testing framework.

⋆ Extensive evaluation of RTS++ on 11 open-source projects, totaling 3,811,916 lines of code and 1,709 test cases.

II. Background and Example

This section provides a brief background on Google Test and illustrates our technique with a simple example.

A. Google Test

Google Test [18] is a popular testing framework for automating test execution for C++ programs. Its design is ispired by JUnit [25], a popular testing framework for Java projects. The terminology used by Google Test is similar to that used by JUnit: a single test function is a *test* and a class containing tests is a *test case*. Google Test supports five different types of tests: Normal Test, Fixture Test, Typed Test, Type-Parameterized Test, and Value-Parameterized Test. RTS++ supports all available test types.

Figure 1a shows an example (`unittest.cc`) that illustrates Normal Tests. The example has three classes under test (`A`, `B`, and `C`) and one test case (`UnitTest`) with three

```
// unittest.cc
class A {
public:
    virtual int foo() {
        return 5; }
};
class B : public A {
public:
    virtual int foo() {
        return 20; }
};
class C: public A {



};

TEST(UnitTest, TestA) {
    A a;
    EXPECT_EQ(5, a.foo());
}
TEST(UnitTest, TestB) {
    B b;
    EXPECT_EQ(20, b.foo());
}
TEST(UnitTest, TestC) {
    C c;
    EXPECT_EQ(5, c.foo());
}
```

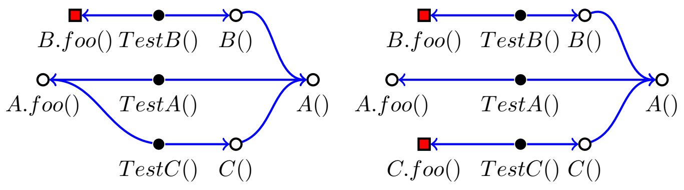
(a) R revision

```
// unittest.cc
class A {
public:
    virtual int foo() {
        return 5; }
};
class B : public A {
public:
    virtual int foo() {
        return 30; }
};
class C: public A {
public:
    virtual int foo() {
    return 30; }
};

TEST(UnitTest, TestA) {
    A a;
    EXPECT_EQ(5, a.foo());
}
TEST(UnitTest, TestB) {
    B b;
    EXPECT_EQ(20, b.foo());
}
TEST(UnitTest, TestC) {
    C c;
    EXPECT_EQ(5, c.foo());
}
```

(b) R' revision

Fig. 1: An illustrative example that shows the old (left) and new (right) revisions of a simple project with three classes and three test cases.



(a) R revision (b) R' revision

Fig. 2: Annotated dependency graphs for the illustrative example. Each node is a function and there is an edge if one function might invoke another function; the shape of the node is squared if the checksum for that node changed between two revisions; newly added nodes are always considered modified.

tests (`TestA`, `TestB`, and `TestC`). `TEST`, similar to `@Test` annotation in JUnit, is a macro used for writing Normal Tests available in the Google Test framework and `EXPECT_EQ` is a macro to assert if an actual value (given as the first argument) matches the expected value (given as the second argument).

B. RTS++ *Illustrated*

Assume, for the sake of the example, that a developer integrates RTS++ at the time of the revision R shown in Figure 1a. If the developer runs tests by running a target from a build script, RTS++ will intercept the execution of the build script after the compilation (and before running tests) and analyze the call graph on LLVM IR. RTS++ will statically build an *annotated dependency graph* as shown in Figure 2a. Test nodes are shown as black nodes and other functions and function members are shown as white circles or red squares. An edge that connects two nodes (n, n') shows that function n *may* invoke function n'. Although the function member `foo` is virtual, generated IR does not use any virtual table because the exact type of each variable and the function member to be invoked is known at the compilation time. In the next section, we describe our support for dynamic dispatch. After the annotated dependency graph is built for the current revision, RTS++ performs the traversal of *both* the *old* graph and the *new* graph to detect affected tests, i.e., those tests whose outcome may be affected by changes. Because this is the first revision of the program that RTS++ analyzed, the old graph does not yet exist and so RTS++ marks all available tests as affected. Additionally, RTS++ stores the new graph for future use.

Assume that the developer modifies the original code to obtain the new code revision R' shown in Figure 1b and runs the tests. RTS++ constructs the new dependency graph shown in Figure 2b. Next, RTS++ traverses both graphs (one at a time) to detect affected tests. It detects modified nodes by comparing the checksums of function bodies between two revisions; these nodes that are modified are marked as red squares in Figure 2. To detect all affected tests, RTS++ computes the upward transitive closure [3] and finds that `TestB` is affected based on the old graph and tests `TestB` and `TestC` are affected based on the new graph. The final result is the union of the two traversals.

The traversal of the new graph was necessary to ensure that we detect those tests that are affected by changes in class hierarchies. While we can often detect the change in a function by observing a change in the metadata in IR, if a function member is added in a subclass and a function that uses that member accepts a pointer to a superclass (e.g., `fun(A *a)`), no changes can be detected; traversing both old and new graphs solves this problem.

III. Technique and Implementation

This section describes RTS++ technique and its implementation. We keep our implementation generic to enable future support for other compilers and testing frameworks. Figure 3 shows a high-level overview of the RTS++ integration into the existing testing processes. RTS++ intercepts the build process, perform analysis to detect affected tests and sends the list of affected tests to the testing framework, which then simply runs those tests.

RTS++ includes three traditional RTS phases, which are all performed statically:

- *Analysis* phase detects the tests that need to be re-executed due to code changes (Section III-A).
- *Execution* phase executes only the selected tests and skips other available tests (Section III-B).
- *Collection* phase saves statically computed metadata needed by the next analysis phase and constructs the annotated program dependency graph (Section III-C).

An *annotated dependency graph* is the key data structure that allows RTS++ to select the tests to be executed. We define it as a tuple $G = (N, E, C, T)$. N is the set of nodes, such

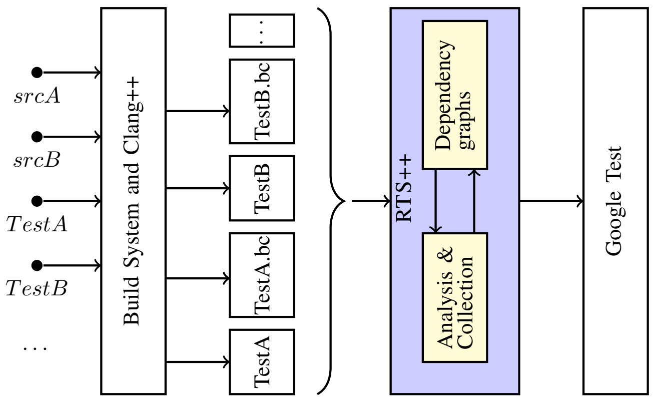


Fig. 3: Overview of the testing process that integrates RTS++. The integration requires minimal changes in the build process. RTS++ processes executable files to obtain a dependency graph for the current project revision, analyzes both old and new dependency graphs to detect affected tests, stores new dependency graph for future use, and sends the list of affected tests to the testing framework.

that each node is a fully qualified name of a function. A fully qualified name includes the namespace, declaring class name, which is optional because a function in C++ need not be inside a class, and the function signature, i.e., function name with argument types. E is the set of edges; an edge connects two nodes iff one function may invoke another function. C is a map from a node to its checksum value, which is computed at the time of the graph construction. Finally, T is the set of tests ($T \subseteq N$).

Next, we describe each phase of RTS++ in more detail. Note that we describe the analysis and collection as separate phases although they are implemented as one pass in our tool; we take this approach to be consistent with prior work on RTS.

A. Analysis

The *analysis phase* identifies tests affected by code changes. RTS++ analyzes all .bc files (i.e., files in LLVM bitstream file format that contains encoding of LLVM IR) in the project. Note that, based on the project configuration, there may be one or more .bc files, and each file may contain one or more tests. Without loss of generality, we describe the analysis performed on a single .bc file that contains multiple tests. Furthermore, we assume that all libraries are statically linked, i.e., analyzed .bc files contains every piece of code that may be needed during test execution. This is achieved by using `-flto` (Link Time Optimization) argument for the linker.

The input to the analysis phase is G, an annotated dependency graph for the *previous* code revision, and the .bc file for the *current* code revision. The graph G is empty when the analysis is invoked for the first time; in this case, all tests discovered in the new graph are considered affected. Otherwise, the graph G is constructed in the collection phase of the previous run (Section III-C).

```
Require: $G$ - graph for the old revision
Require: $bc$ - bitcode file for the current/new revision
 1: function ANALYSIS($G$, $bc$)
 2:     $G' \leftarrow$ CONSTRUCTCALLGRAPH($bc$)
 3:     $affected \leftarrow \emptyset$
 4:     $affected \mathrel{+}=$ FINDAFFECTED$(G.E, G.C)$
 5:     $affected \mathrel{+}=$ FINDAFFECTED$(G'.E, G.C)$
 6:     return $\{\tau \mid \tau \in affected \land \tau \in G'.T\}$
 7: end function

Require: $edges$ - edges in the graph being traversed revision
Require: $checksums$ - map from each function to its checksum in
    the old revision
 8: function FINDAFFECTED($edges$, $checksums$)
 9:     $affected' \leftarrow \emptyset$
10:     repeat
11:         $affected \leftarrow affected'$
12:         for all $(f, f') \in edges$ do
13:             if CKFUN$(f') \neq checksums[f'] \lor f' \in affected$ then
14:                 $affected' \mathrel{+}= \{f, f'\}$
15:             end if
16:         end for
17:     until $|affected| == |affected'|$
18:     return $affected'$
19: end function
```

Fig. 4: Analysis phase in RTS++. The algorithm shown is intentionally kept simple (and inefficient) to simply the presentation in the paper.

Figure 4 shows the algorithm for the analysis phase. First, on line 2, RTS++ constructs an annotated dependency graph for the new version of the executable. Second (line 3), it defines an empty set that will represent a set of all affected functions. A function is affected if its checksum is modified; we assume the existence of a function CKFUN that takes a function as an input and computes a checksum of its body.

Third, the algorithm invokes the `FindAffected` function on both the old graph and the new graph. The reason to traverse both graphs is to ensure that tests affected due to changes in the class hierarchy are included in the set of affected tests; the *old graph* is traversed to detect those tests that are affected by a *removal* of a virtual function member from a class hierarchy, and the *new graph* is traversed to detect those tests that are affected by an *addition* of a virtual function member in a class hierarchy. Finally, on line 6, the algorithm extracts only tests available in the new revision from the set of impacted functions, which is the final set of selected tests.

`FindAffected` function traverses the edges of the graph and for each callee, it computes its new checksum value by invoking the CKFUN function. If the checksum is different from the time when the graph was constructed or the function is in the set of affected nodes, then both the caller and callee are added to the set of affected nodes. The loop stops when it reaches the fixed-point, i.e., the set of affected nodes does not change.

We implemented the analysis phase as an LLVM Pass. Specifically, we leverage the LLVM's `CallGraphSCCPass` pass, which allows us to traverse the call graph of an executable [8]. The main function in the pass that we override, `runOnSCC`, runs on every node in a call graph. One of the reasons for our design decision to traverse the old and new graphs separately (rather than simultaneously, as it was done for dynamic RTS with control-flow graphs [34], [38]) was the existence of the `CallGraphSCCPass` pass. We leverage this pass to obtain an efficient and robust implementation, which follows from well-tested LLVM code.

B. Execution

The *execution phase* runs the selected tests discovered during analysis. Integration of RTS++ into the well established testing process is straightforward.

In C++ projects, it is common to have a target in a build script for running tests. Projects that use Google Test simply invoke (one or more) executables that contain tests, e.g., `./unittests`. Rather than running the executable, we perform four steps. First, we obtain the list of available tests in the executable by executing `./unittests --gtest list tests`. This list is needed to be able to filter only some tests for the execution as described below. Second, we invoke RTS++'s analysis phase to select tests. Third, we map names of selected tests to the names used by Google Test obtain in the first step. Finally, we invoke the original executable and pass the list of tests to execute via `./unittests --gtest filter=colon_separated_list_of_tests`.

An implementation challenge in the execution phase is keeping track of the test names between the source code (which we refer to as *filter test names*) and the names Google Test uses in the executable (which we refer to as the *compiled test names*). Table I shows compiled and filter test names for various test types. Name matching for Normal Test or a Fixture Test is straightforward. Next, for Typed Test, we match the types and construct the filter test name. For Type-Parameterized Test, we overapproximate the set of tests by using global prefix, e.g., `--gtest filter=*TypeParamTest/0.Test1`. Finally, we check if a test is Value-Parameterized Test, and if so, we run the test with generic prefix and suffix (e.g., `--gtest filter=*Value- ParamTest.Test1/*`).

C. Collection

The *collection phase* prepares the dependency graph used in the current and next run of the analysis phase. Recall that the analysis and collection phases are implemented as one pass in RTS++. In this section, we describe the way we build and annotate the graph; the algorithm is shown in Figure 5.

First, we build a class hierarchy by processing .bc files. Second, we traverse the call graph to save dependencies between functions. If a call is non-virtual, we simply store that the caller function depends on the callee function. In the case of a virtual call, we find the base class (let's call it `C`) based on the call and hierarchy that we built. Then, we traverse all classes in the hierarchy of `C` (both superclasses and subclasses). If a class (let's call it `B`) in the hierarchy has a function member with the same signature as the callee (the SIG function obtains a signature for a given function), we store the

TABLE I: "Filter" and "Compiled" Test Names; Filter Format is Used by Google Test to Exclude Tests from Execution and Compiled Format is Available in Executable Files.

Test Type	Filter Name	Compiled Name
Normal Test	TestCase.TestName	TestCase_TestName_Test::TestBody()
Fixture Test	FixtureTest.TestName	FixtureTest_TestName_Test::TestBody()
Typed Test	TypedTest/0.Test1 # TypeParam = A	TypedTest_Test1_Test<A>::TestBody()
Type-Parameterized Test	Prefix/TypeParamTest/0.Test1 # TypeParam = A	gtest_case_TypeParamTest::Test1<A>::TestBody()
Value-Parameterized Test	Prefix/ValueParamTest.Test1/0	ValueParamTest_Test1_Test::TestBody()

```
Require: $bc$ - bitcode file
1: function CONSTRUCTCALLGRAPH($bc$)
2:     $N \leftarrow E \leftarrow C \leftarrow T \leftarrow \emptyset$
3:     $inheritance \leftarrow \emptyset$
4:     for all $c \in$ EXTRACTCLASSES($bc$) do
5:         for all $s \in$ GETSUPERCLASSES($c$) do
6:             $inheritance \mathrel{+}= (c, s)$
7:         end for
8:     end for
9:     for all $c \in$ EXTRACTCLASSES($bc$) do
10:        for all $m \in$ GETMETHODS($c$) do
11:            $N \mathrel{+}= \{m\}$
12:            if ISTEST($m$) then
13:                $T \mathrel{+}= \{m\}$
14:            end if
15:            $cksum \leftarrow 0$
16:            for all $i \in$ GETINSTRUCTIONS($m$) do
17:                $cksum \mathrel{+}=$ CKFUN($i$)
18:                if ISFUNCTIONCALL($i$) then
19:                    $callee \leftarrow$ EXTRACTCALLEE($i$)
20:                    ▷ for non-virtual calls
21:                    $E \mathrel{+}= \{(m, callee)\}$
22:                    ▷ for virtual calls
23:                    $base \leftarrow$ BASECLS(DECLCLS($callee$))
24:                    $E \mathrel{+}= \{(m, m') \mid m' ismemberof\ c'$
25:                        $\wedge (c', base) \in inheritance$
26:                        $\wedge$ SIG($m'$) == SIG($callee$)$\}$
27:                end if
28:            end for
29:            $C \mathrel{+}= \{(m, cksum)\}$
30:        end for
31:    end for
32:    return $(N, E, C, T)$
33: end function
```

Fig. 5: Collection phase in RTS++.

dependency between the caller and the called function member in class B. However, we do not store dependencies on classes that do not implement the virtual function member; keeping such dependencies are unnecessary because we traverse both the old and the new graphs during the analysis phase.

To find the base class for a virtual function call we analyze `getelementptr` instruction, which contains the index into a virtual table. This is one of the places where we depend on a specific ABI, specifically the Itanium C++ ABI.

During the graph traversal, we also compute the checksum of each function; we create a mapping from each function to its checksum. We extend the `FunctionComparator` class available in LLVM to compute SHA-1 checksum based on instruction opcodes, constant values, and global values used as operands of instructions.

TABLE II: Supported Change Types.

Name	Description
DI	Delete an instance initializer
AI	Add an instance initializer
CI	Change an instance initializer
DSM	Delete a static non-initializer method
ASM	Add a static non-initializer method
CSM	Change a static non-initializer method
DIM	Delete an instance non-initializer method
AIM	Add an instance non-initializer method
CIM	Change an instance non-initializer method
DGF	Delete a global function
AGD	Add a global function
MGF	Modify a global function

D. Build System Integration

RTS++ can be easily integrated with AutoMake, CMake, and Make with minimal changes to the build scripts by using the GNU Gold Linker [17], which is distributed with the newer versions of the GNU Bintools, and link time optimization (LTO). To enable link time optimization, the flag `-flto` should be passed to both the compiler and the linker. Additionally, the option `save-temps` should be passed to the linker to emit the bitcode for the executable.

E. Safety

RTS++ is *safe*, i.e., it guarantees to select tests affected by changes, for any code change except those related to non-primitive global variables, function pointers, and use of `setjump` and `longjump`; if such code is present, a user has to decide if RTS++ should be used. The safety of RTS++ follows from the safety of RTS techniques based on control-flow dependencies, which was proven in the past [38]. Table II shows the list of supported change types, i.e., types of changes for which RTS++ can detect affected tests. We leave it as future work to formally prove the safety of RTS++.

F. Generalizability

Among the three phases – analysis, execution, and collection – two of them – analysis and collection – are independent on any testing framework. Our implementation follows a modular design, such that each phase can easily be replaced to support a new testing framework, different hashing function, new graph traversal, etc.

IV. EVALUATION

Our evaluation answers the following research questions:

RQ1: What is the reduction in the number of executed tests obtained by RTS++ compared to retest-all across many revisions of popular open-source projects?

TABLE III: Subjects Used in Our Evaluation.

Project	URL	#Revs.	SHA	LOC	Binary (MB)	BuildSys	#Tests	#Test Cases
Abseil	https://github.com/abseil/abseil-cpp	14	9c9477fa	53,468	100	CMake	1,084	207
Boringssl	https://github.com/google/boringssl.git	50	9b2c6a9	193,262	56	CMake	839	57
gRPC	https://github.com/grpc/grpc.git	50	17f682d	1,406,407	1,934	CMake	1,682	150
Kokkos	https://github.com/kokkos/kokkos.git	50	d3a9419	128,452	45	Make	227	29
Libcouchbase	https://github.com/couchbase/libcouchbase.git	50	b028b9e	98,931	12	CMake	303	62
Libtins	https://github.com/mfontanini/libtins.git	50	b18c2ce	92,707	285	CMake	797	63
OpenCV	https://github.com/opencv/opencv.git	33	9a8a964	1,018,274	670	CMake	21,106	673
Protobuf	https://github.com/google/protobuf.git	50	264e615	317,477	116	AutoMake	2,086	193
Rapidjson	https://github.com/Tencent/rapidjson.git	34	af223d4	82,536	46	CMake	425	33
Rocksdb	https://github.com/facebook/rocksdb.git	31	2c2f388	238,995	13,886	Make	2,571	208
Tiny-dnn	https://github.com/tiny-dnn/tiny-dnn.git	45	1c52594	181,407	9	CMake	294	34
Total	N/A	407	N/A	3,811,916	17,159	N/A	31,414	1709
Average	N/A	37	N/A	346,538	1,560	N/A	2,856	155

RQ2: What is the reduction in the end-to-end testing time obtained by RTS++ compared to retest-all across many revisions of popular open-source projects?

Prior to answering these research questions (Section IV-D), we describe the system configuration (Section IV-A), the subjects used in our experiments (Section IV-B) and experiment setup (Section IV-C).

A. System Configuration

We ran experiments on a 4-core 3.9 GHz AMD 1800X CPU with 8GB of RAM running Ubuntu Linux 16.04 LTS. We used Clang++ and LLVM 6 as well as Google Test 1.8. Some projects bundled their own version of Google Test, which was either 1.7 or 1.8. We ran experiments several times and observed only minor noise in execution time across multiple runs of experiments.

B. Subjects

As stated earlier, there has been no recent work on regression testing for C++ projects. Therefore, we had to find projects that would be appropriate to be used for evaluating our technique/implementation. Thus, our *minor contribution* is the list of projects and buildable revisions that could be used for evaluating regression testing techniques for C++. We followed steps taken by recent research on regression testing for Java: we searched for popular projects (in terms of the number of stars) on GitHub. Additionally, we filtered out those projects that do not use AutoMake/CMake/Make or Google Test. We also ignored those projects that we were not able to build successfully on our platforms; we dedicated up to 8h for setting up each project.

Table III shows the final set of projects used in our evaluation. For each project we show its name, URL, number of buildable revisions (in the latest 50 revisions), latest SHA at the time of our experiments, number of lines of code (LOC), size of the binary file, build system used, number of tests, and number of test cases; we defined "test" and "test case" in Section II. The last five columns are computed for the latest revision. We can see that the projects differ in size (between 53,468 and 1,406,407), build system, and number of tests (between 227 and 21,106).

The last two rows show, when applicable, the total and average values across all projects. In sum, we used 11 projects, totaling 3,811,916 lines of code and 31,414 tests.

Abseil is a collection of C++ code that extends the C++ standard library; this project migrated to CMake recently, which is the reason we use only a small number of revisions. Boringssl is Google's fork of OpenSSL. gRPC is a modern open-source remote procedure call (RPC) framework developed by Google. Kokkos is a programming model in C++ that provides abstractions for code to run efficiently on different hardware such as multi-core CPUs and GPUs. Libcouchbase is the C client for Couchbase (an open-source NoSQL database). Libtins is a high-level C++ library for sending, receiving, and manipulating network packets. OpenCV stands for Open Source Computer Vision Library. Protobuf is a language and platform agnostic method for serializing data. Rapidjson is a C++ library for parsing and generating JavaScript Object Notation (JSON). Rocksdb is an embedded database for key-value storage developed by Facebook. Tiny-dnn is a header-only deep learning library written in C++. In the end, we were able to build most of revisions for most of the projects, and some failing builds were expected as building old revisions leads to a number of challenges [46].

C. Experiment Setup

Our experiment setup closely follows recent work on RTS for Java, which performed extensive evaluations using open-source projects and large number of revisions [16], [27], [35], [51]; this evaluation approach started with work on Chianti. Our setup only differs in the details due to the differences in the technology used by C++ projects, e.g., integration with a build system, obtaining the list of available tests, etc. Specifically, we perform the following steps for each subject (P) in Table III.

a) `clone(`P`,`P`.SHA)` - clone the latest revision (as specified in Table III) of the project,
b) `checkout(`P`, -50)` - checkout a revision from the history that is 50 commits prior to the latest revision. If a project does not have 50 revisions, checkout the first revision available in the repo,

TABLE IV: Test Selection Results Using RTS++.

"#Tests retest-all" - total number of tests across all revisions executed, "#Tests RTS++" - total number of tests executed with RTS++ across all revisions, "Time retest-all" - total time to execute all tests, "Time RTS++" - total time to execute tests with RTS++, "Ratio test" - ratio of executed tests retest-all/ RTS++ * 100, "Ratio time" - ratio of execution time retest-all/ RTS++ * 100.

Project	#Tests		Time [s]		Ratio [%]	
	retest-all	RTS++	retest-all	RTS++	test	time
Abseil	15,176	1,312	2,583	617	8.65	23.90
Boringssl	41,523	6,756	1,606	478	16.27	29.76
gRPC	84,085	4,797	31,637	17,511	5.70	55.35
Kokkos	5,609	1,251	13,540	3,614	22.30	26.69
Libcouchbase	14,668	1,403	1,494	178	9.57	11.91
Libtins	38,922	1,607	43	1,334	4.13	3,083.39
OpenCV	695,505	206,804	11,621	4,364	29.73	37.55
Protobuf	104,300	2,918	796	2,357	2.80	296.13
Rapidjson	14,349	1,546	929	805	10.77	86.66
Rocksdb	78,390	6,715	39,517	25,487	8.57	64.50
Tiny-dnn	13,034	525	48,977	5,849	4.03	11.94
Total	1,105,561	235,634	152,743	62,594	-	-
Average	-	-	-	-	11.14	38.70*

*the average ratio for time excludes Libtins and Protobuf; these projects are the smallest in terms of the test execution time and analysis and collections phases dominate the test execution phase.

c) `build(P)` - build the current revision. If the build (of at least one module) is successful proceed to the next step; otherwise go to step f),
d) `retest-all(P)` - run all available tests,
e) `RTS++(P)` - integrate RTS++ into the project, analyze the project, execute selected tests, and collect new meta-data that will be used in the next analysis run,
f) if this is the latest revision, stop the loop; otherwise checkout the subsequent revision and go to step c).

During the execution of steps d) and e), we collect execution logs that we post-process to extract data necessary to answer our research questions. From step d), we extract the number of available tests ($N_{retest-all}$) and total testing time ($T_{retest-all}$). From step e), we extract the number of selected tests (N_{RTS++}) and *end-to-end testing time* (T_{RTS++}); end-to-end testing time includes all phases of RTS++: analysis, test filtering, execution, and collection.

Based on the extracted data, we compute two key metrics used to evaluate the benefits of RTS++. (1) Test selection ratio ($N_{reduction}$), which is computed as a ratio of the number of selected tests over the total number of available tests, i.e., $N_{reduction}= N_{RTS++}/ N_{retest-all}*$ 100. (2) Time ratio ($T_{reduction}$), which is computed as a ratio of end-to-end time taken by RTS++ and testing time for retest-all, i.e., $T_{reduction}= T_{RTS++}/ T_{retest-all}*$ 100. Finally we compute the average values for both $N_{reduction}$ and $T_{reduction}$ across all revisions used in the experiments.

D. Results

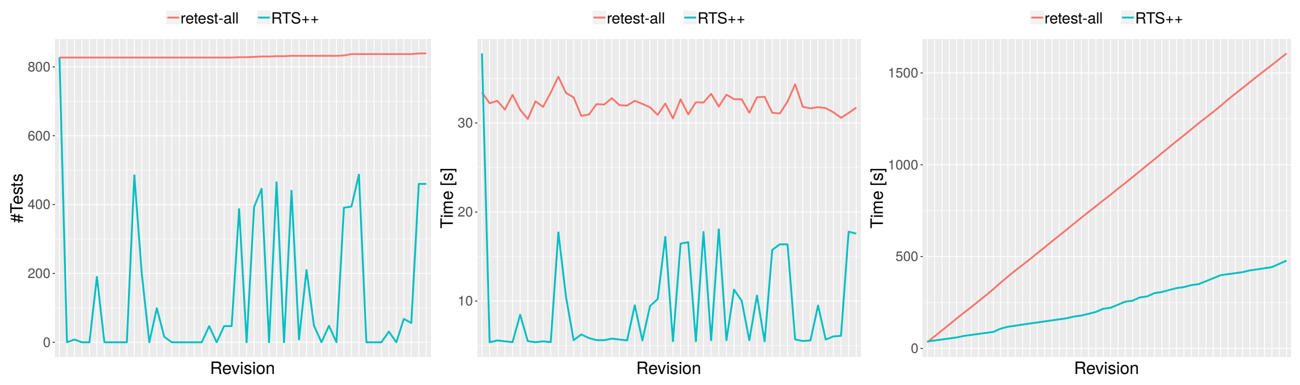
a) **Reduction in the Number of Executed Tests:** This section answers RQ1. Table IV summarizes the number of executed tests for all projects used in the study. Column 2 shows the total number across all revisions of executed tests with retest-all, and Column 3 shows the total number of selected tests with RTS++. Column 6 shows test selection ratio ($N_{reduction}$) averaged across all revisions. We can see that test selection ratio varies from 2.80% (for Protobuf) to 29.73% (for OpenCV).

The last two rows of the table show total and average values, across all projects, if applicable. It is worth noting that the total number of tests executed by retest-all and RTS++ was 1,105,561 and 235,634, respectively.

b) **Reduction in the End-to-End Test Execution Time:** This section answers RQ2. Table IV also shows end-to-end testing time for all projects used in the study. Column 4 show the total time (across all revisions) to run tests with retest-all. Column 5 shows total end-to-end testing time for RTS++. Column 7 shows time ratio ($T_{reduction}$) averaged across all revisions. The results show that RTS++ substantially reduces time for all but two projects. These two projects, Libtins and Protobuf are the *smallest* among all projects used in our study in terms of the test execution time, and RTS++ analysis and collection phases take more time than the test execution phase; this is expected for projects with short test time, and it is a reasonable expectation that these projects would not benefit from any RTS technique [16].

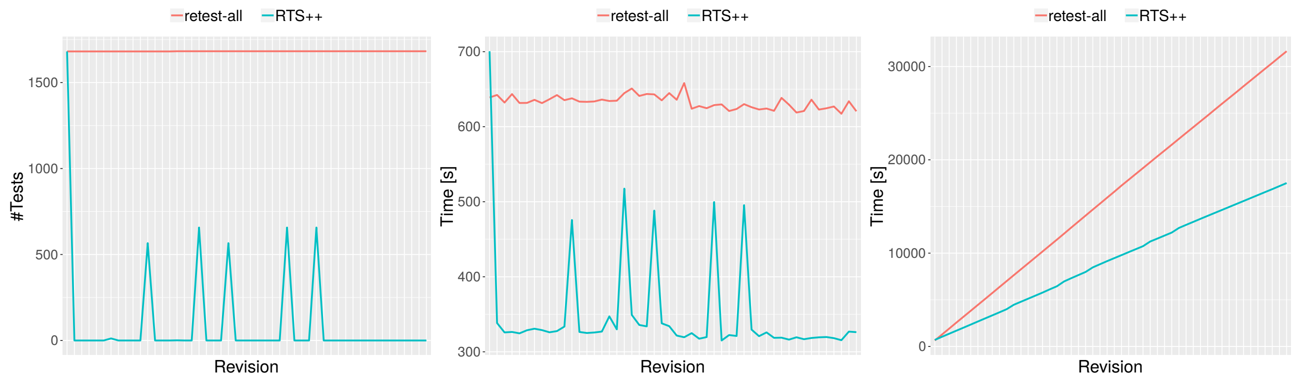
As before, the last two rows of the table show total and average values, across all projects if applicable. It is worth noting that the total time when using retest-all and RTS++ was 152,743 seconds and 62,594 seconds, respectively.

c) **Representative Case Studies:** Figures 6 and 7 visualize the raw data extracted from the execution logs, as described in the previous section, for Boringssl and gRPC, respectively. Each figure has tree subfigures: (a) number of tests executed with retest-all and RTS++, (b) end-to-end testing time for



(a) Number of executed tests (b) End-to-end testing time (c) Cumulative end-to-end testing time

Fig. 6: Results for Boringssl.



(a) Number of executed tests (b) End-to-end testing time (c) Cumulative end-to-end testing time

Fig. 7: Results for gRPC.

retest-all and RTS++, and (c) cumulative end-to-end testing time over all revisions used in our experiments. We do not show plots for other projects due to the space limits.

We can observe (figures 6a and 7a) that at no revision, RTS++ selects all the tests. As expected, and one of the assertions in our scripts, the number of tests run by retest-all and RTS++ has to be the same for the first revisions used in the experiments.

Furthermore, we can observe (figures 6b and 7b) that the end-to-end testing time for RTS++ is longer than testing time for retest-all for the first revision; this is expected as all tests are executed *and* RTS++ collects metadata, i.e., checksums and the annotated call graph.

Finally, we can observe (figures 6c and 7c) that RTS++ can save substantial time even when only a subset of software history is considered.

V. Discussion

Build time vs. test time. For the sake of completeness, we report and compare build time (excluding test time) and test time for projects used in our experiments. Figure 8 shows, for each project, build time (blue) and test time (green). Note that we use the log scale for y-axis. The results showed in this figure are only for the latest revisions of the projects. Although testing may not always be the dominant part of a build, we focus on speeding up test execution, as recent work on incremental build systems [1], [3] does not support fine-grained test selection and testing still takes substantial time.

Java vs. C++. Recent work on RTS for Java has shown that two language features – dynamic class loading and reflection – may compromise the safety of static RTS techniques [27]. A static RTS technique for C++ does not face those issues because C++ does not support reflection and projects frequently statically link libraries for the testing purpose, i.e., the entire call graph is known. We also note that there was no need to implement the smart checksum [27], i.e., the checksum that ignores debug info not observed by tests, e.g., original source code lines. Clang strips those debug info automatically unless a developer specifies the O0 optimization level; none of the projects used in our study uses O0.

Note that RTS++ supports various C++ language features, including preprocessor macros and templates. This is supported by design because we analyze executable file, where macros and templates are not available any longer.

Test order dependencies. Dependencies among tests (e.g., via shared/static variables) may cause problems for an RTS tool [4], [19], [30], [31]. If a tool selects only dependent test, the test may fail as it will start in an unexpected state. If such cases exist, a developer can specify that some tests have to be run prior to some other tests.

Zero selected tests. We have observed that RTS++ selects no

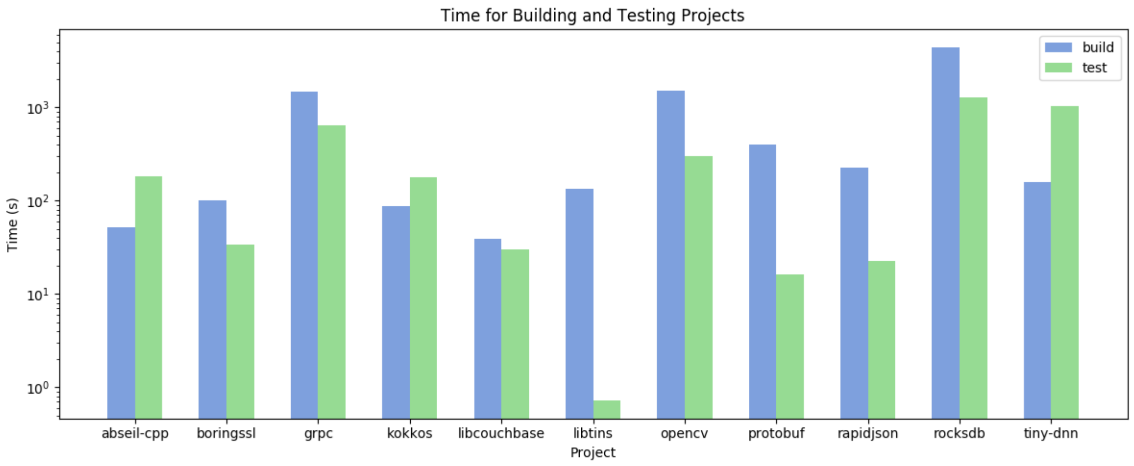


Fig. 8: Build vs. test time for projects used in the experiments.

test at many revisions (e.g., Figure 7b); similar observations were made for Java projects [16]. Our closer look at several revisions revealed that developers frequently change other files that do not impact tests, e.g., README. However, further research is needed to understand various types of changes and their potential impact on an RTS tool [9].

Future work. Although our experiments showed that RTS++ can be useful for large projects that have long testing time, several other questions remain. (1) Improve precision by removing dependency on classes that are not instantiated by a test. Namely, if a test only instantiates classes `A` and `B` (such that `B` extends `A`), we do not need to add dependencies on other classes that extend class `A`. We can detect this statically by finding the set of constructors potentially invoked from the test. (2) Improve precision by dynamically collecting dependencies. We plan to explore using LLVM XRay instrumentation tool [49] to collect dynamic dependencies. It will be interesting to compare differences in precision and overhead for static and dynamic approaches. (3) Support the Boost testing framework [6] and integrate RTS++ with the Bazel build system [3]. (4) Develop adaptive [2] and hybrid [52] RTS techniques for C++.

VI. Threats To Validity

Our work is subject to several threats, similar to prior work on regression testing and RTS.

External. We used 11 projects in our evaluation, but these projects may not be representative of all C++ projects. To mitigate this threat, we searched for popular open-source projects on GitHub developed by large software organizations, which differ in size and the application domain.

We ran our experiments on (max) 50 revisions per project. Had we used a different number of revisions or different time frame, we could have observed different results. We chose the number of revisions per project in line with recent work on RTS; looking at longer sequences leads to broken builds due to changes in compiler version, build system, etc. [46].

The reported results are obtained on a single platform. However, our findings could be different if we run experiments on a different platform. To mitigate this threat, we did run a subset of experiments on another platform; while the absolute numbers are not exactly the same across platforms, the benefits of using RTS++ are clear.

Internal. Our implementation and scripts may contain bugs, which may impact our findings. To mitigate this threat we performed three-fold testing approach. First, we wrote a number of unit tests for our tool, which is a common practice in industry. Second, two of the authors manually inspected more than 200 revisions in the projects used in our study and agreed if tests should be selected or not; manually finding the exact set of tests that should be executed is infeasible. In the future, we plan to check safety, precision, and generality of RTS++ by adopting the RTSCheck framework [53] to C/C++. Finally, the scripts for running the experiments were built on top of the infrastructure used by other researchers.

Our implementation currently does not detect function invocations via pointers. This requires more engineering effort, and considering that function pointers are not widely used in C++, we plan to support this in the future.

Construct. Although several RTS techniques have been proposed for C/C++, to the best of our knowledge, there are no publicly available tools that implement these techniques. As mentioned earlier in this paper, there has been no active research on RTS for C/C++ for over a decade, so even if some tools were publicly available they would not be readily applicable to projects that use LLVM and Google Test.

VII. Related Work

RTS has been studied for decades, but it remains a relevant topic [22], [33]; several surveys nicely summarize prior work on RTS [5], [14], [50]. In this section we briefly present most closely related work and contrast the prior work with the work presented in this paper.

Early techniques supported projects written in C/C++ [10], [26], [37], [39]. Rothermel and Harrold [37] presented a technique (for C) that builds control dependency graphs to detect affected tests; this work was evaluated in later years [38]. In their work, dependencies for tests were collected dynamically. Chen et al. [10] developed a technique that combines static and dynamic analysis for C; dynamic part was used to capture function invocations via pointers. Kung et al. [26] introduced class firewall approach to detect affected tests by tracking dependencies among classes. Rothermel et al. [39] were among the first to study RTS for C++ and object-oriented languages in general. Their work uses interprocedural control flow graph to select tests, analyzes source code of two project revisions, and uses dynamic coverage. Our work targets C++ and uses call graph analysis to detect affected tests. RTS++, unlike prior work, supports projects that compile to LLVM bitcode and use Google Test. Moreover, we evaluated our technique on actual revisions from publicly available open-source projects rather than on project releases.

Ren et al. [35] developed Chianti, a tool that selects affected tests by detecting the impact of various atomic changes on a call graph. Chianti was developed for Java, analyzes source code to detect atomic changes, uses dynamic call graphs, and is evaluated only on a single project. Jang et al. [24] developed an approach similar to Chianti for C++; their tool was evaluated on a single small program (26 classes) and revisions were manually created by the authors. Orso et al. [34] presented a hierarchical RTS technique. In the first phase – partitioning – their technique finds what classes and interfaces are changed (and those that depend on the changed classes). In the second phase, the technique analyzes only classes that belong to the partition and detects dangerous edges [38]. Their approach requires traversal of two – old and new – Java interclass graphs (JIG) simultaneously. Additionally, their approach assumes that dynamic coverage information is readily available. Arguably, RTS++ is most closely related to Chianti and the work on dangerous edges. Unlike prior work, RTS++ targets C++, does not analyze source code but binaries, and does not assume availability of any dynamically computed coverage but computes coverage statically. RTS++ is also evaluated on much larger set of projects.

Gligoric et al. [16] presented Ekstazi, an RTS technique based on dynamic class-dependencies for Java projects. Ekstazi has been adopted by both open-source projects and industry. Ekstazi# [47] implements the Ekstazi technique for .NET platform. Legunsen et al. [27] developed and evaluated STARTS, an RTS technique based on the class firewall. STARTS was extensively evaluated, and the results showed that STARTS compares favorably with Ekstazi. Zhang [51] presented a hybrid technique that dynamically tracks dependencies on both methods and classes. Recently, Wang et al. [48] introduced the first refactoring-aware RTS, i.e., an RTS technique that does not run tests that are affected only by behavior-preserving transformations. Celik et al. [9] presented RTSLinux, an RTS technique that tracks dynamic dependencies for tests running in a JVM that spawns other processes or uses native code. Like these recent RTS projects, RTS++ is also extensively evaluated using open-source projects and large number of revisions available in public repositories of those projects. Unlikely recent work, RTS++ targets C/C++ projects that compile to LLVM bitcode and use Google Test. Additionally, RTS++ tracks dependencies on fine-grained level (i.e., functions); interestingly, recent work on Java showed that using fine-grained dependencies (i.e., methods) may lead to high overhead [16], [27], [51], and our findings in this paper show that function-level granularity can provide substantial reduction in test execution time for C++ projects. RTSLinux would behave as retest-all for C/C++ code that compiles to a single executable.

An interesting future direction is to develop a coarse-grained RTS technique for C++ by extending RTS++ and compare its performance with the technique presented in this paper. Romano et al. [36] introduced an exotic technique, named SPIRITuS, which uses information retrieval to perform an unsafe RTS for Java projects. It would be interesting to implement SPIRITuS for C++ and compare the performance with RTS++.

RTS++ is also related to other work on regression testing, such as test case prioritization, i.e., ordering tests with the goal to run the failing tests earlier (e.g., [12], [40], [42]), and test-suite reduction, i.e., detecting likely duplicate tests without significant reduction in the fault detection capability (e.g., [20], [32], [41]). Our infrastructure could be used as a good starting point for (re)implementing or evaluating test case prioritization and test-suite reduction techniques for C++.

Finally, our work is related to (modern) build systems, e.g., Bazel [3], Buck [1], and CloudMake [11], which commonly keep an explicit list of dependencies/files for each build target. RTS++ performs more precise analysis as it keeps dependencies for individual tests and tracks dependencies on function members rather than files. Thus, RTS++ can be integrated in projects that use these modern build systems to improve their precision.

VIII. Conclusion

We presented a novel RTS technique, named RTS++, that targets projects written in C++, which use the LLVM IR and the Google Test testing framework. RTS++ implements an RTS technique based on static function-level call graph analysis; to ensure correctness in the presence of inheritance, RTS++ analyzes call graphs obtained from two revisions. RTS++ integrates with many existing build systems, including AutoMake, CMake, and Make. RTS++ was evaluated on 11 large open-source projects, totaling 3,811,916 lines of code and 1,709 test cases. We measured the benefits of RTS++ compared to running all available tests (i.e., retest-all) in terms of the number of executed tests, as well as end-to-end testing time. Our results show that RTS++ reduces the number of executed tests and end-to-end testing time by up to 97.20% and 88.09%, respectively. Based on the results presented in this paper, RTS++ can be a valuable addition to any large C++ project that uses continuous integration system, i.e., runs tests frequently, and has tests that run for long time.

Acknowledgments. We thank the anonymous reviewers for their comments; Ahmet Celik, Christine Julien, Karl Palmskog, Pengyu Nie, and Chenguang Zhu for their feedback on this work. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363, CCF-1652517, CCF-1703637 and CCF-1704790.

References

[1] Buck. http://buckbuild.com.

[2] Md. Junaid Arafeen and Hyunsook Do. Adaptive regression testing strategy: An empirical study. In *International Symposium on Software Reliability Engineering*, pages 130–139, 2011.

[3] Bazel. https://bazel.build.

[4] Jonathan Bell and Gail E. Kaiser. Unit test virtualization with VMVM. In *International Conference on Software Engineering*, pages 550–561, 2014.

[5] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.

[6] Boost.Test. https://www.boost.org/doc/libs/1_67_0/libs/test/doc/html/index.html.

[7] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: an industrial case study. In *Symposium on the Foundations of Software Engineering, formal tool demonstrations*, pages 975–980, 2016.

[8] CallGraphSCCPass. http://llvm.org/doxygen/classllvm_1_1CallGraphSCCPass.html.

[9] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. Regression test selection across JVM boundaries. In *International Symposium on Foundations of Software Engineering*, pages 809–820, 2017.

[10] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. TestTube: A system for selective regression testing. In *International Conference on Software Engineering*, pages 211–220, 1994.

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

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

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

[14] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Journal of Information and Software Technology*, 52(1):14–30, 2010.

[15] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's distributed and caching build service. In *International Conference on Software Engineering, Software Engineering in Practice*, pages 11–20, 2016.

[16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[17] Gold linker. https://en.wikipedia.org/wiki/Gold_(linker).

[18] Google Test. https://github.com/google/googletest.

[19] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*, pages 223–233, 2015.

[20] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *International Conference on Software Engineering*, pages 738–748, 2012.

[21] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *International Working Conference on Source Code Analysis and Manipulation*, 2018.

[22] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *International Working Conference on Source Code Analysis and Manipulation*, pages 117–132, 2018.

[23] Jean Hartmann. Applying selective revalidation techniques at Microsoft. In *Pacific Northwest Software Quality Conference*, pages 255–265, 2007.

[24] Y. K. Jang, M. Munro, and Y. R. Kwon. An improved method of selecting regression tests for C++ programs. *Journal of Software Maintenance*, 13(5):331–350, 2001.

[25] JUnit. https://junit.org/junit4.

[26] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[27] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.

[28] LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[29] Writing an LLVM Pass. http://llvm.org/docs/WritingAnLLVMPass.html.

[30] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[31] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *International Symposium on Foundations of Software Engineering*, pages 496–499, 2011.

[32] Voas JM. Offutt J, Pan J. Procedures for reducing the size of coverage-based test sets. In *International Conference on Testing Computer Software*, pages 111–123, 1995.

[33] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Future of Software Engineering*, pages 117–132, 2014.

[34] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*, pages 241–251, 2004.

[35] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.

[36] Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. SPIRITuS: A simple information retrieval regression test selection approach. *Information and Software Technology*, 99:62–80, 2018.

[37] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. In *International Conference on Software Maintenance*, pages 358–367, 1993.

[38] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[39] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[40] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *International Conference on Software Maintenance*, pages 179–188, 1999.

[41] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *International Symposium on Foundations of Software Engineering*, pages 237–247, 2015.

[42] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *International Symposium on Software Testing and Analysis*, pages 97–106, 2002.

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

[44] Tools for continuous integration at Google scale. http://www.youtube.com/watch?v=b52aXZ2yi08.

[45] Travis CI. https://travis-ci.com.

[46] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 2017.

[47] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. File-level vs. module-level regression test selection for .NET. In *Symposium on the Foundations of Software Engineering, industry track*, pages 848–853, 2017.

[48] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. Towards refactoring-aware regression test selection. In *International Conference on Software Engineering*, 2018. 233–244.

[49] XRay. https://llvm.org/docs/XRay.html.

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

[51] Lingming Zhang. Hybrid regression test selection. In *International Conference on Software Engineering*, pages 199–209, 2018.

[52] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *International Conference on Software Maintenance*, pages 23–32, 2011.

[53] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. A framework for checking regression test selection tools. In *International Conference on Software Engineering*, 2019. To appear.