The Thesis committee for Ben Fu
Certifies that this is the approved version of the following thesis:

Regression Test Selection for C++ Based on Call Graph Analysis

APPROVED BY

SUPERVISING COMMITTEE:

Assistant Professor Milos Gligoric, Supervisor

Professor Christine Julien

Regression Test Selection for C++ Based on Call Graph Analysis

by

Ben Fu, B.S.E.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2018

Dedicated to my parents and my fiancée.

Acknowledgments

I would like to first thank my thesis advisor, Assistant Professor Milos Gligoric at The University of Texas at Austin for his guidance and expertise. He was extremely knowledgeable on the topic and displayed endless amounts of patience and wisdom at every step of the way.

I would also like to thank Professor Christine Julien for being the second reader for my thesis and dedicating her time to provide invaluable feedback for me.

Finally, I am thankful for my parents and my fiancée for constantly supporting me throughout my work on this thesis and all of my years of study at The University of Texas at Austin. This thesis would not have been possible without them.

Regression Test Selection for C++ Based on Call Graph Analysis

Ben Fu, M.S.E.
The University of Texas at Austin, 2018

Supervisor: Assistant Professor Milos Gligoric

Regression testing – running available tests after each project change – is widely practiced in industry to check that a project change does not break working functionalities. Despite its widespread use and importance, regression testing is a costly activity, and the cost is steadily increasing with the increase in the number of tests and the number of changes. Regression test selection (RTS) attempts to optimize regression testing activity by deselecting tests not affected by project changes and executing remaining tests. RTS has been extensively studied and several tools have been deployed in industry. However, work on RTS over the last decade has mostly focused on managed languages (e.g., Java). Meanwhile development practices (e.g., frequency of commits, testing framework, compilers, etc.) in C++ projects have dramatically changed, and the way we should design and implement RTS tools and the benefits of those tools is unknown.

This thesis presents a design and implementation of an RTS technique, named Ekstazi++, that targets projects written in C++, which use the LLVM compiler and the Google Test testing framework. Ekstazi++ implements an RTS technique based on the call graph analysis. Ekstazi++ integrates with many existing build systems, including AutoMake, CMake, and Make. We evaluated Ekstazi++ on 11 large open-source projects, totaling 3,811,916 lines of code. We measured the benefits of Ekstazi++ 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 Ekstazi++ reduces the number of executed tests and end-to-end testing time up to 97.20% and 88.09%, respectively.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Regression testing – running available tests at each project revision to check correctness of recent project changes – is widely practiced in industry. Widespread availability of continuous integration systems simplified build configurations to enable regression testing, and continuous integration services, e.g., TravisCI [29], provide necessary resources even to open-source projects. Although regression testing is extremely important, it is rather a costly activity, and this cost tends to increase with the increase in the number of tests and the frequency of project changes [27, 28]. Many large software organizations, including Apache, Google, Microsoft, Salesforce, and Uber have reported high costs of regression testing and have been adopting various techniques to reduce this cost [3, 5, 7, 12, 26].

Regression test selection (RTS) attempts to optimize the regression testing activity by *deselecting* tests that are *not* anticipated to be affected by recent project changes and running only the remaining subset of tests [1, 6, 25, 33]. Traditionally, coverage based RTS techniques keep a mapping from each test to all code elements (e.g., statements, basic blocks, functions, etc.) that the test might be using and deselect those tests (at a new project revision)

that do not depend on any modified code elements. The mapping from each test to code elements can be obtained either *statically* (without running the test) or *dynamically* (during test execution). 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 (not-so-recent) surveys summarize RTS status and progress [1, 6, 33]. Researchers (and practitioners) have studied RTS techniques for various programming languages, including C/C++ (e.g., [4]), C# (e.g., [30]), and Java (e.g., [8, 15, 21, 22, 35]), kept dependencies on various code elements, such as basic blocks (e.g., [26]), functions (e.g., [4]), and classes (e.g., [8]), and used both static (e.g., [15]) and dynamic (e.g., [8, 34, 35]) analyses.

Motivation. Most of the initial work on RTS techniques was (not surprisingly) focused on languages that compile to unmanaged code, e.g., C, but, interestingly, work in the last decade has mostly focused on RTS for managed languages, such as Java and C#. Thus, the impact of the revolution of C++ compilers (e.g., increasing popularity of LLVM), testing frameworks (e.g., widespread use of Google Test), and development processes (popularity of GitHub and continuous integration services) on RTS *design, implementation, and provided benefits* is unknown. Additionally, recent practices to evaluate RTS implementations based on *end-to-end* execution or test time have not been used to evaluate RTS tools that target C++ projects.

Technique. We designed, developed, and evaluated a novel RTS technique

named Ekstazi++ that supports projects that compile to LLVM IR [16], use Google Test, and follow modern development practices. Ekstazi++ implements an RTS technique based on call graph analysis, i.e., it keeps a mapping from each test to (transitively reachable) functions that are potentially used by the test. At a new project revision, Ekstazi++ analyzes code changes and runs those tests that depend on one of the modified/deleted/added functions. Ekstazi++ ensures selection of the appropriate set of tests even in the presence of a dynamic dispatch by analyzing the call graphs for both old and new project revisions. Ekstazi++ can be seen as a revival of RTS techniques that analyze control-flow graphs and dangerous edges [13, 21, 22, 25].

Implementation. We implemented Ekstazi++ as an LLVM compiler pass [17]. The benefit of this approach is that Ekstazi++ can be extended in the future to support other languages that compile to LLVM bitcode. Ekstazi++ blends nicely with Google Test and supports all test types (e.g., Typed Tests, Type-Parameterized Tests, Value-Parameterized Tests, etc.). However, Ekstazi++ is in no way dependent on Google Test and could easily be integrated with other testing frameworks, such as the Boost Test Library [2]. Ekstazi++ currently works with the AutoMake, Make, and CMake build systems.

Evaluation. We performed an extensive evaluation of Ekstazi++ on 11 open-source projects available on GitHub, totaling 3,811,916 lines of code and 1,709 test cases. We used (up to) the 50 latest revisions of each project. To assess the benefits of Ekstazi++, we measured savings compared to *retest-all* (i.e., running all tests at each project revision) in terms of the number of ex-

ecuted tests and end-to-end testing time. Our results show that Ekstazi++ can provide substantial savings. In terms of the number of tests, Ekstazi++ led to a reduction of up to 97.20% compared to retest-all. In terms of end-to-end testing time, Ekstazi++ led to a reduction of up to 88.09% compared to retest-all.

Chapter 2

Example and Background

This section introduces Ekstazi++ through an example and introduces basic terminology related to RTS.

2.1 Illustrative Example

To best walk through an example using Ekstazi++, we first show the source code for an example unit test in C++. Then, we show how to retrieve the metadata needed for test selection, which consists of the set of function hashes and the dependency graph for the test code. Finally, we show how to utilize the metadata to select specific tests to run using the popular C++ testing framework, Google Test [10].

After walking through the initial example, we make a small modification to the source code and then repeat the steps above for the modified version of the example.

2.1.1 Source Code

To best explain the design and process of Ekstazi++, we walk through a simple example shown in Listing 2.1. In this example, we have three classes:

`A`, `B`, and `C`, and three test cases: `UnitTest.TestA`, `UnitTest.TestB`, and `UnitTest.TestC`. Classes `B` and `C` both inherit from class `A`, and `B` overrides the method `foo()` while `C` does not. The test `UnitTest.TestA` tests the method `A::foo()`, the test `UnitTest.TestB` tests the method `B::foo()`, and the test `UnitTest.TestC` tests the method `C::foo()`.

Listing 2.1: Example test code.

```
1
2 // unittest.cc
3 class A {
4 public:
5     virtual int foo() {
6         return 5;
7     }
8 };
9
10 class B : public A {
11 public:
12     virtual int foo() {
13         return 20;
14     }
15 };
16
17 class C: public A {
18
19 };
20
21 TEST(UnitTest, TestA) {
22     A a;
23     EXPECT_EQ(5, a.foo());
24 }
25
26 TEST(UnitTest, TestB) {
27     B b;
28     EXPECT_EQ(20, b.foo());
29 }
30
31 TEST(UnitTest, TestC) {
32     C c;
```

```
33      EXPECT_EQ(5, c.foo());
34  }
```

The `TEST()` macro simply defines a Google Test test function, and the parameters to the macro are the *test case* name and the *test* name respectively[1]. Thus, in this example, we have one test case, `UnitTest`, with three tests: `TestA`, `TestB`, and `TestC`.

2.1.2 Function Hashes

We calculate a unique hash value for each function in the code so that we will know in the future if the function is modified or not. The process of calculating a hash value for each function is similar to existing RTS approaches such as Ekstazi, which calculates a hash value for each class [8]. In this case, we compute the hash values for the functions `A::foo()`, `B::foo()`, `UnitTest_TestA_Test::TestBody()`, `UnitTest_TestB_Test::TestBody()`, and `UnitTest_TestC_Test::TestBody()`. Note that we do *not* calculate the hash value for `C::foo()` since the class `C` did not override the `foo()` function. To calculate the hash value for a function, we walk through the function's statements and hash the instructions, operands, and constant values. We store the name of each function and its hash value in a metadata file with the name suffix `functions.txt`. The exact algorithm we use to compute the unique hash value of a function is described later in detail. For our illustrative example, the function hashes metadata file is shown in Listing 2.2.

[1]Note that a test case corresponds to a test class and a test corresponds to a test method.

Listing 2.2: Function hashes for the illustrative example.

```
// unittest.functions.txt

A::foo();6293160677188140734
B::foo();6651799985476877791
UnitTest_TestA_Test::TestBody();15250928343042435800
UnitTest_TestB_Test::TestBody();16493337799901246779
UnitTest_TestC_Test::TestBody();2227623096648085615
```

2.1.3 Dependency Graph

Another piece of metadata that we collect is the dependency graph for the test executable; multiple test cases (and tests) can be in a single executable. The dependency graph is constructed by observing the function calls that occur throughout the code, i.e., call graph analysis. It is represented as a directed graph in which the nodes represent functions and the edges represent a depended-on-by relationship. A visual representation of the dependency graph for our illustrative example is shown in Figure 2.1.

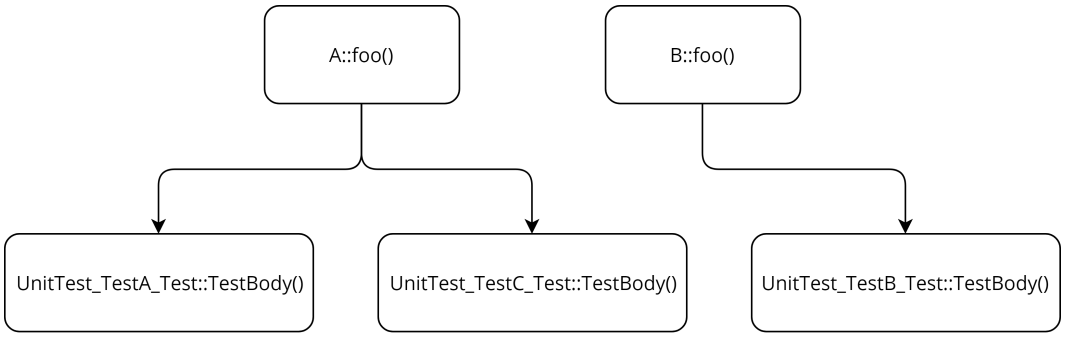


Figure 2.1: Dependency graph for the illustrative example.

To store this graph into a metadata file, we save the graph as an adjacency list representation in a file with the suffix `depgraph.txt`. The format

of the metadata file is shown in Listing 2.3.

Listing 2.3: Dependency graph output for the illustrative example.

```
A::foo();UnitTest_TestA_Test::TestBody();UnitTest_TestC_Test::TestBody()
B::foo();UnitTest_TestB_Test::TestBody()
```

The function `A::foo()` is depended on by both `UnitTest.TestA` and `UnitTest.TestC` because it is called by both tests. Likewise, the function `B::foo()` is depended on by `UnitTest.TestB`.

2.1.4 Analysis and Execution

Before discussing the process of test selection, it is important to understand how the source code is compiled to an executable. In this case, we assume the source file `unittest.cc` is compiled to a single executable named `unittest`. Running the executable `./unittest` would run all tests.

Since this is the initial run of Ekstazi++, there is only one version of the metadata that has been generated. All tests are treated as modified since they are newly added and did not exist before. Thus, we initially select all of the tests to run.

To run a subset of tests in the test executable, we run the executable with the flag `--gtest_filter` set to the tests we would like to select. Since this is the initial run, we will select all three tests, `UnitTest.TestA`, `UnitTest-.TestB` and `UnitTest.TestC`. The `--gtest_filter` flag accepts strings separated with ":" to select tests where the strings are the full names of the tests (`TestCaseName.TestName`). Thus, the full command to select and run all of

the tests is `./unittest --gtest_filter=UnitTest.TestB:UnitTest.TestC`. The output of this command is shown in Listing 2.4.

Listing 2.4: Test output when running the command `./unittest --gtest_filter=UnitTest.TestB:UnitTest.TestC`.

```
Running main() from gtest_main.cc
Note: Google Test filter = UnitTest.TestB:UnitTest.TestC:UnitTest.TestA:
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from UnitTest
[ RUN ] UnitTest.TestA
[ OK ] UnitTest.TestA (0 ms)
[ RUN ] UnitTest.TestB
[ OK ] UnitTest.TestB (0 ms)
[ RUN ] UnitTest.TestC
[ OK ] UnitTest.TestC (0 ms)
[----------] 3 tests from UnitTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[ PASSED ] 3 tests.
```

As expected, all of the tests are executed since all three tests were passed to the Google Test filter. In the following section, we make a simple modification to the source code and walk through the process of selecting the tests that should be run.

Note that listing all of the tests after the `--gtest_filter` flag is the same as simply running the test executable with no flag. However, for the purposes of consistency, Ekstazi++ always uses the `--gtest_filter` flag whether or not all of the tests are selected.

2.2 Modified Version of the Illustrative Example

In the previous section we showed the process of collecting metadata from the test program. Now, the code will be modified with two changes: the function `B::foo()` will return 10 instead of 20, and we will add a new function `C::foo()`, which returns the value 30 (and overrides `A::foo()`). Logically, this should cause the tests `UnitTest.TestB` and `UnitTest.TestC` to be run again, and both tests should fail since the expected values from the tests are no longer valid. `UnitTest.TestA` should not be rerun since it is not affected by the changes, i.e., behavior remains the same.

2.2.1 Source Code

The source code for the modified version is shown in Listing 2.5.

Listing 2.5: Same test code but with `B::foo()` modified and `C::foo()` added. The changes are marked in green.

```
1  // unittest.cc
2
3  class A {
4  public:
5      virtual int foo() {
6          return 5;
7      }
8  };
9
10 class B : public A {
11 public:
12     virtual int foo() {
13         return 10;
14     }
15 };
16
17 class C: public A {
18 public:
```

```
19      virtual int foo() {
20        return 30;
21      }
22 };
23
24 TEST(UnitTest, TestA) {
25     A a;
26     EXPECT_EQ(5, a.foo());
27 }
28
29 TEST(UnitTest, TestB) {
30     B b;
31     EXPECT_EQ(20, b.foo());
32 }
33
34 TEST(UnitTest, TestC) {
35     C c;
36     EXPECT_EQ(5, c.foo());
37 }
```

2.2.2 Function Hashes

Since the function `B::foo()` was modified, the new calculated hash for the function is different. Additionally, the new function `C::foo()` is introduced and its hash value is calculated. The new function hash metadata file is shown in Listing 2.6.

Listing 2.6: Function hashes for modified example.

```
// unittest.functions.txt

A::foo();6293160677188140734
B::foo();9228047671960258583
C::foo();5563320140227721397
UnitTest_TestA_Test::TestBody();15250928343042435800
UnitTest_TestB_Test::TestBody();16493337799901246779
UnitTest_TestC_Test::TestBody();2227623096648085615
```

Comparing this set of function hashes to those of the original example,

we see that the hash value for `B::foo()` is different and there is an additional function, `C::foo()`. None of the other functions were modified, so their hash values did not change.

2.2.3 Dependency Graph

The dependency graph is now modified because `UnitTest.C` no longer depends on the function `A::foo()` but rather the new function `C::foo()`. The dependency graph for the modified example is illustrated in Figure 2.2.

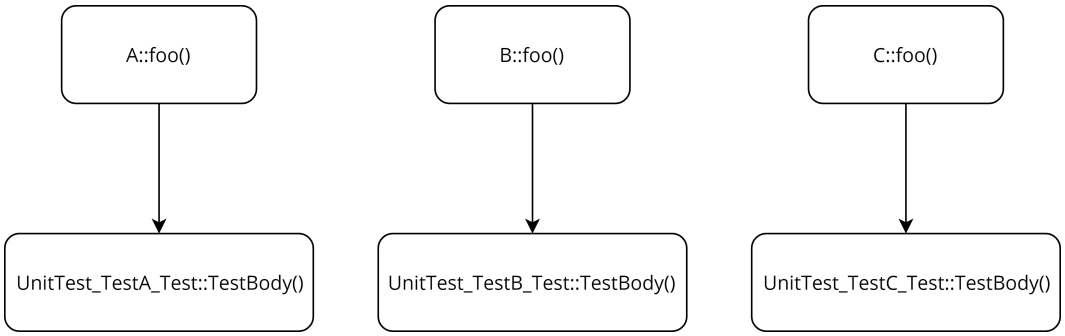


Figure 2.2: Dependency graph for modified example.

As with before, we save the adjacency list representation of this dependency graph to a metadata file with the suffix `depgraph.txt`. The file's contents are shown in Listing 2.7.

Listing 2.7: Dependency graph file for modified example.

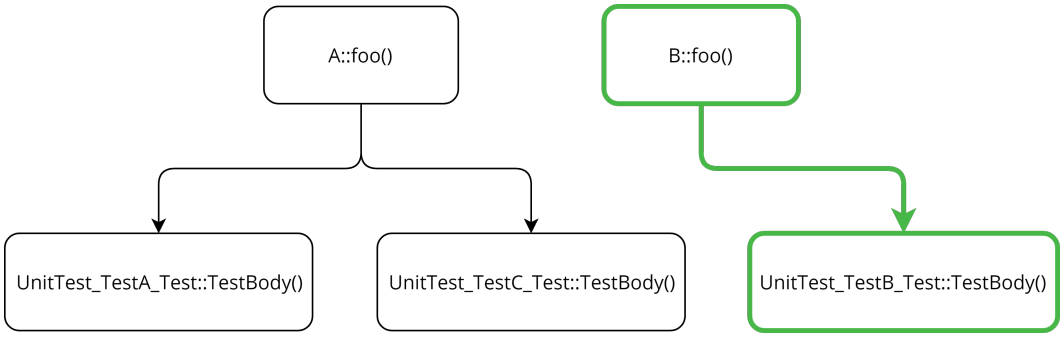
```
// unittest.depgraph.txt

A::foo();UnitTest_TestA_Test::TestBody()
B::foo();UnitTest_TestB_Test::TestBody()
C::foo();UnitTest_TestC_Test::TestBody()
```

2.2.4 Analysis and Execution

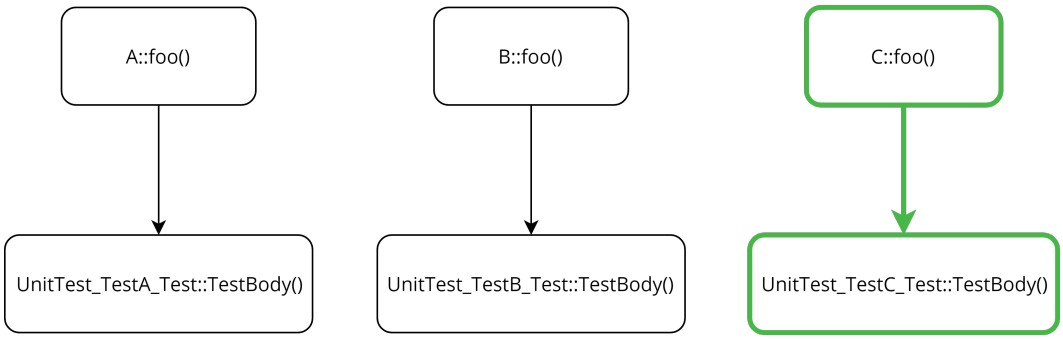
To select the tests that should be run after changing the source code, we use the information we know from the saved metadata. We now know that the function `B::foo()` has been modified since it has a different hash value than before, and we also know that `UnitTest.TestB` depends on `B::foo()` from the (old) dependency graph. Additionally, we know that the function `C::foo()` is new, and `UnitTest.TestC` now depends on `C::foo()`. `A::foo()` has not been modified since it has the same hash value as before.

We first look at the modified function `B::foo()`. To find all tests that are affected by the change to `B::foo()`, we perform a search in the original dependency graph starting from `B::foo()`. We find that `UnitTest_TestB_Test-::TestBody()` is encountered during traversal, so we know that the test `Unit-Test.TestB` must be rerun. No other test function was encountered, so no other tests were affected by the change. Now, we observe the new function `C::foo()`. We first look in the old dependency graph for `C::foo()` but it does not exist. Thus, we look in the new dependency graph and find that `C::foo()` does exist. Now, we perform a graph traversal from `C::foo()` and find that the test function `UnitTest_TestC_Test::TestBody()` is encountered, so the test `UnitTest.TestC` should be rerun. The visualization of this process is shown in Figure 2.3.

Now that we know that `UnitTest.TestB` and `UnitTest.TestC` need to be rerun, we know the filter string to pass to the Google Test filter flag. As with before, we run the executable with the `--gtest_filter` flag set, but now



(a) Dependency graph traversal for original example from `B::foo()` with the traversal path in green.



(b) Dependency graph traversal for modified example from `C::foo()` with the traversal path in green.

Figure 2.3: Traversing the dependency graphs after modifying `B::foo()` and adding `C::foo()`.

we set the flag to `UnitTest.TestB:UnitTest.TestC` since we only want to run those tests. Thus, the complete command to run the test is `./unittest --gtest_filter=UnitTest.TestB:UnitTest.TestC`. The output for selecting and running these tests is shown in Listing 2.8.

Listing 2.8: Test output when running `./unittest --gtest_filter =UnitTest.TestB:UnitTest.TestC`.

```
Running main() from gtest_main.cc
Note: Google Test filter = UnitTest.TestB:UnitTest.TestC:
[==========] Running 2 tests from 1 test case.
```

```
[----------] Global test environment set-up.
[----------] 2 tests from UnitTest
[ RUN ] UnitTest.TestB
ekstazipp/examples/tests/method-inh-add-modify/v2-gtest.cpp:32: Failure
Expected equality of these values:
  20
  b.foo()
    Which is: 10
[ FAILED ] UnitTest.TestB (0 ms)
[ RUN ] UnitTest.TestC
ekstazipp/examples/tests/method-inh-add-modify/v2-gtest.cpp:37: Failure
Expected equality of these values:
  5
  c.foo()
    Which is: 30
[ FAILED ] UnitTest.TestC (0 ms)
[----------] 2 tests from UnitTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 2 tests, listed below:
[ FAILED ] UnitTest.TestB
[ FAILED ] UnitTest.TestC

  2 FAILED TESTS
```

We can see that both `UnitTest.TestB` and `UnitTest.TestC` were run, and both tests failed since the expected results are now different from the actual results. This section gave a quick overview of the process of Ekstazi++ by walking through an example. In the next section, we describe the design and implementation of Ekstazi++ as well as go over the process introduced in this section in more detail.

Chapter 3

Design and Implementation

3.1 Architecture

The basic system architecture consists of the Ekstazi++ LLVM Pass and the Ekstazi++ Test Runner. We first compile the source code using one of the supported build systems into LLVM Intermediate Representation (IR) and test executables. For each test executable in the project, we first run the Ekstazi++ LLVM Pass on the executable's LLVM IR. The Pass will collect metadata information including the unique hash of each function, the dependency graph corresponding to the test executable, and the set of modified tests. Then, we run the test executables and pass the corresponding flags to select a subset of the tests using the Ekstazi++ Test Runner. The illustration of this process can be found in Figure 3.1.

3.1.1 Collected Metadata

In order to perform the test selection, we need to know which tests were modified between two project revisions due to changes in the source code. To compute the modified tests, we store two types of metadata when running the Ekstazi++ LLVM Pass: the unique function hash of each function in the executable and the dependency graph for the test executable. From this

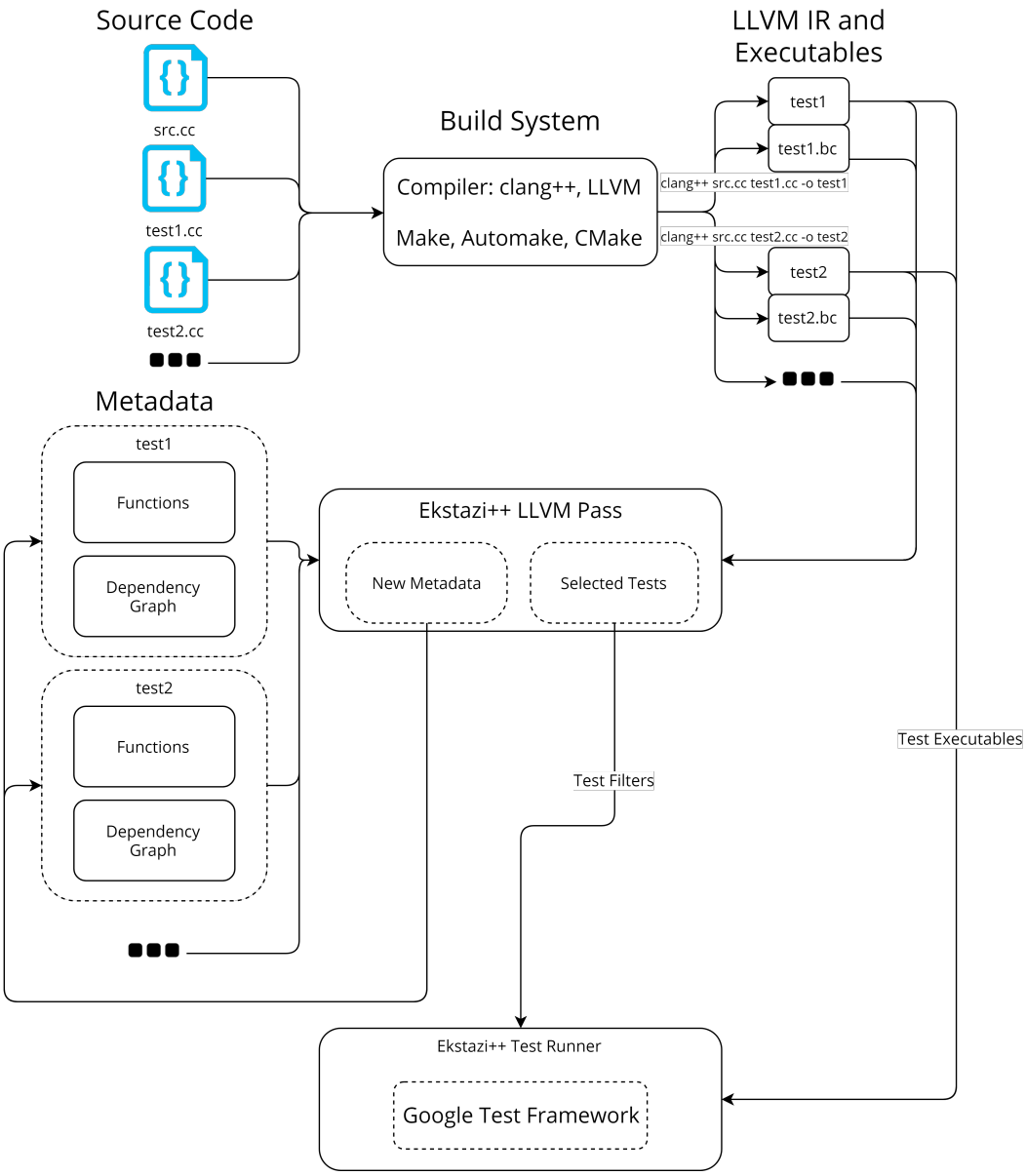


Figure 3.1: Block diagram that illustrates the integration of Ekstazi++ into the workflow.

metadata, we can compute the list of modified functions and subsequently the list of modified tests.

3.1.1.1 Detecting Modified Functions

To determine which functions have changed in the code, we compute a unique hash value for each function based on its bitcode structure. When the function's source code is modified, its hash value will also change. Thus, we can retrieve all the functions that have been modified by comparing the hash value of the function in an old revision of the code to that of the newer revision of the code. A function is defined as *modified* if it is added in the new revision of the code, if it is removed in the new revision of the code, or if it exists in both revisions of the code but has a different hash value in the new revision.

3.1.1.2 Dependency Graph

After detecting which functions have changed, we need to compute all the transitive dependencies for the modified functions. To accomplish this, we build a dependency graph unique to *each test executable*. That is, two different test executables will have different dependency graphs, even though they may both link to the same static library. The dependency graph nodes represent functions in the code, and the edges represent dependencies, i.e., call relation. To compute all the transitive dependencies, we start at the nodes corresponding to each modified function and then traverse the graph until we

reach all the leaf nodes. An example of the dependency graph metadata can be found in Listing 2.3.

With the exception of the first run, each run of the Ekstazi++ LLVM Pass has access to two dependency graphs: the dependency graph belonging to the previous revision of the code and the dependency graph belonging to the current revision. After computing which functions have been modified from the previous revision by comparing their new hash values with their old hash values, we traverse both the old dependency graph *and* the new dependency graph starting from the modified functions in order to find which test functions have been modified. At first, it appears that traversing the old dependency graph alone is enough to compute the modified functions because intuitively, a function's dependencies cannot change if the function body itself does not change. This is true for most cases. However, in the case of inheritance, it is possible for a function's dependencies to change without a change in the code if a new derived function is added (refer back to the example in the previous chapter). Similarly, it is not enough to simply traverse the new dependency graph alone because if an inherited method is removed, we still need the old dependency graph to detect the dependencies of the removed function which could potentially be a test. Therefore, the solution to find all possible dependencies affected by a change is to traverse both the old dependency graph and the new dependency graph.

3.2 Test Selection

To perform test selection, we use the metadata presented in the previous section to first compute which test functions were affected by changes to the source code. Then, we pass these modified test functions to the test framework that will select and run the tests. Currently, Ekstazi++ supports the Google Test framework. To select tests in Google Test, we need to run the test executable with the flag `--gtest_filter` and pass the names of the tests we would like to run.

3.3 Algorithm

An algorithmic representation of the test selection process can be found in Listing 3.1. In the algorithm, *test module* refers to the LLVM IR that corresponds to the test executable being analyzed.

Listing 3.1: Algorithm for test selection.

```
1 function select_and_run(test_module, old_function_hashes, old_depgraph):
2   type_hierarchy = graph()
3   vtables = map(type, vtable)
4   depgraph = graph()
5   function_hashes = map(function_name, hash_value)
6
7   for each type in test_module.type_metadata:
8     add (type, supertype) relationships to type_hierarchy
9
10  for each vtable in test_module.vtables:
11    add (type, vtable) to vtables
12
13  for each call_site in test_module.call_graph:
14    if call_site is direct:
15      compute hashes for caller and callee
16      add caller and callee to function_hashes
```

```
17        add (caller, callee) to depgraph
18
19      else: // call site is virtual
20        get pointer_type from call_site
21        get vtable_offset from call_site
22        get derived types for pointer_type from type_hierarchy
23        for each type in derived types:
24          get vtable for type from the vtables
25          get callee from vtable using vtable_offset
26          compute hashes for caller and callee
27          add caller and callee to function_hashes
28          add (caller, callee) to depgraph
29
30    get modified_functions by computing set union of old_function_set and
          function_set
31    for each function in modified_functions:
32      traverse old_depgraph and new_depgraph to find all dependent functions
33      if dependent function is a test:
34        add function to set of modified tests
35
36    for each test in set of modified tests:
37      compute test_filter for test
38      add test_filter to test_filters
39
40    for each test in all_tests:
41      if test matches a a test filter from test_filters:
42        run test
```

3.4 Ekstazi++ LLVM Pass

The Ekstazi++ dependency analysis is written as a LLVM Pass to be integrated with the LLVM compiler infrastructure. Currently, the pass can run on compiled LLVM Intermediate Representation files (.bc and .ll). The pass implements LLVM's `CallGraphSCCPass`, which allows us to traverse the call graph of the program. The main pass function, `runOnSCC()`, runs on every call graph node in the program being compiled. Each call graph node represents

a function, and each call graph edge represents a function call.

3.4.1 Inputs

The only input to the Ekstazi++ LLVM Pass is the name of the test executable. The test executable is necessary to probe all of the tests. For example, for Google Test, running the executable with the flag `--gtest_list_tests` will not actually run the tests but instead list all of the tests that would be run when running the executable. Most of the time, the test executable will have the same name of the actual bitcode file (e.g., test1 will have a bitcode file named test1.bc). However, not all bitcode files exist right next to their corresponding executable. It is common in modern projects to copy the executable to a different folder at the end of the build process. Thus, there is no way of knowing where the test executable exists from the LLVM Pass itself, so we pass in an argument that is the full path to the executable that corresponds to the bitcode file.

3.4.2 Key Steps

We describe the key steps in more detail.

3.4.2.1 Initialization

Before the main Pass function runs, the Ekstazi++ LLVM Pass first loads the information, i.e., metadata, from the previous run into memory. If this is the first run of Ekstazi++, then the function hashes, dependency graph,

and tests will all be empty. The Pass then collects all of the test information. In the case of Google Test, Ekstazi++ runs the test executable with the flag `--gtest_list_tests` and then parses the output of the command to gather all of the test cases and test names.

Type Hierarchy During Pass initialization, we build the type hierarchy for the module to store all types and inheritance relationships in the module. To build the type hierarchy, we parse the type metadata found in the module. The LLVM IR for the module contains the type metadata for each defined class or struct in the code as well as its polymorphic relationships. For example, for the code in Listing 3.2, the corresponding type metadata is shown in Listing 3.3.

Listing 3.2: Example code showing simple inheritance relationships. In this example, `A` is a declared classes that does not derive from any other class. Class `B` and `C` both derive from class `A`.

```
1 class A {
2   virtual void foo();
3 };
4
5 class B : A {
6   virtual void foo();
7 };
8
9 class C : A {
10  virtual void foo();
11  virtual void bar();
12 };
```

Listing 3.3: Type metadata in LLVM IR for the example in Listing 3.2.

```
1 @_ZTV1A = constant [...], !type !0
2 @_ZTV1B = constant [...], !type !0, !type !1
3 @_ZTV1C = constant [...], !type !0, !type !2
4
```

```
5 !0 = !{i64 16, !"_ZTS1A"}
6 !1 = !{i64 16, !"_ZTS1B"}
7 !2 = !{i64 16, !"_ZTS1C"}
```

The mangled string `_ZTV1` indicates that the global constant is a virtual function table for the corresponding class (e.g., `_ZTV1A` corresponds to the virtual function table for the class `A`).

Virtual Function Tables To identify virtual function tables, we observe the global constants defined in the LLVM IR that corresponds to the module. Since both Clang and GCC follow the standard CXX ABI for name mangling, we can identify which constants are virtual function tables by looking at the names of the constants. The following listing shows an example of what the virtual function table is for the example we described in Section 2.

Listing 3.4: Virtual function table constants in LLVM IR.

```
1 @_ZTV3A = internal dso_local unnamed_addr constant { [3 x i8*] } { [3 x i8*] [i8
      * null, i8* bitcast ({ i8*, i8* }* @_ZTI3A to i8*), i8* bitcast (i32 (%class.A*)
      * @_ZN3A4fooEv to i8*)] }, align 8, !type !6
2 @_ZTV3B = internal dso_local unnamed_addr constant { [3 x i8*] } { [3 x i8*] [i8
      * null, i8* bitcast ({ i8*, i8* }* @_ZTI3B to i8*), i8* bitcast (i32 (%class.B*)
      * @_ZN3A4fooEv to i8*)] }, align 8, !type !6
3 @_ZTV3C = internal dso_local unnamed_addr constant { [3 x i8*] } { [3 x i8*] [i8
      * null, i8* bitcast ({ i8*, i8* }* @_ZTI3C to i8*), i8* bitcast (i32 (%class.C*)
      * @_ZN3A4fooEv to i8*)] }, align 8, !type !6
```

The CXX ABI defines the format and elements of the virtual function table. The first element is the offset, which is used for multiple inheritance. The second element is the run-time type information, or RTTI, which is used to expose the type information of an object at runtime. The third element is the first virtual function pointer, and the subsequent elements correspond to

the next virtual function pointers. In our illustrative example, we have only one virtual function, which is the function `foo()`. In the new revision of the example, both classes `B` and `C` override `A::foo()`. Thus, the third element of class `A`'s virtual function table is the virtual function pointer that points to `A::foo()`. Similarly, the third element of `B`'s virtual function table is the pointer to `B::foo()`, and the third element of `C`'s virtual function table is the pointer to `C::foo()`.

3.4.2.2 Function Hashing

As we traverse the call graph of the program, we obtain a unique hash for each encountered function by taking into account the structure of its basic blocks. To hash the value of the function, we walk through each of its basic blocks and analyze its instructions.

Parameters that we include in the hashing algorithm include the order of instructions, type of instructions, and compile-time value of the operands of the instructions. The function hash algorithm is derived from the LLVM Function Hash algorithm defined in the `FunctionComparator` class, which computes a hash value for a function based on the types of instructions, order of instructions, and number of operands for each instruction. We extended this algorithm by also hashing the known compile-time values of the operands of instructions such as global variables.

Basic Block Traversal A basic block is defined as a section of code with a single entry and a single exit. Each function has one or more basic blocks, and each basic block is made up of instructions. For every basic block in a function, we iterate over its instructions. Each instruction has an opcode and operands. To compute a unique hash for the instruction, we hash the type of instruction (the opcode), the order of the operands, and the values of the operands if retrievable.

Value Hashing Constant values such as globals can be hashed by retrieving their initial values that are available at the time of compilation. To hash values, we first test if a specific value is a constant value. If it is a constant, then we hash its value.

3.4.2.3 Dependency Graph Building

For each call graph node, we query every other node that it is connected to. The edges to those nodes represent function calls from the current node. Thus, we can build a dependency graph for each test module by traversing the LLVM Call Graph in the `CallGraphSCCPass`.

LLVM defines two types of function calls: direct and indirect. In direct calls, both the caller and callee are clearly defined functions. That is, there is no ambiguity regarding which function is called. In this case, we can simply add the call to our dependency graph since we know the exact names of the functions (and classes) for both the caller and the callee. Indirect calls are

challenging because although the caller is clearly defined, the callee is not. In the C++ language, function pointers and virtual function calls both fall into the category of indirect calls. In these cases, there are multiple candidates for callees. We currently support virtual function calls in Ekstazi++ but not function pointer calls.

Virtual Function Calls A virtual call is represented in LLVM IR as a series of instructions that first loads the virtual function table from the class, retrieves the virtual function pointer located at the specified virtual table offset, and then makes an indirect function call using the virtual function pointer. This series of instructions that corresponds to the example in section 2 can be found in Listings 3.5 and 3.6.

Listing 3.5: The virtual function call in source code from the example in Listing 2.1.

```
1 A* b = new B();
2 EXPECT_EQ(20, b->foo());
```

Listing 3.6: Sequence of instructions in LLVM IR that correspond to the virtual function call in Listing 3.5.

```
1 %15 = bitcast %class.B* %13 to %class.A*
2 %16 = load %class.A*, %class.A** %3, align 8
3 %17 = bitcast %class.A* %16 to i32 (%class.A*)***
4 %18 = load i32 (%class.A*)**, i32 (%class.A*)*** %17, align 8
5 %19 = getelementptr inbounds i32 (%class.A*)*, i32 (%class.A*)** %18, i64 0
6 %20 = load i32 (%class.A*)*, i32 (%class.A*)** %19, align 8
7 %21 = call i32 %20(%class.A* %16)
```

In the Ekstazi++ LLVM Pass, we know whether a function call is direct or indirect. To retrieve the actual function being called, we backtrack

from the `call` instruction until we are at the `getelementptr` instruction. The operands to this instruction are the type of the object's pointer as well as the index to retrieve the object from. In this case, the type of the pointer is class `A`, specifically the virtual function table of `A`, and the index into the virtual function table is 0. It is important to note that this index is *not* the actual index into the virtual table but rather the index into the virtual table starting from the beginning of the virtual functions. Recall that the first element of the virtual table is always the offset from the top of the class, and the second element of the virtual table is the runtime type information (RTTI). The first virtual function is actually located at the third index of the virtual table. Thus, there is always a difference of 2 between the *real* index into the virtual table and the index that is from the `getlementptr` instruction. In our case, the index 0 means that the real index we want to retrieve from the vtable is 2. The algorithm for retrieving the virtual function call can be found in Listing 3.7.

Listing 3.7: Algorithm for retrieving the actual function data from a virtual function call.

```
1 function add_dependencies_from_vfcall(call_site):
2   find getelementptr instruction before call_site
3   get class_type from first operand to getelementptr instruction
4   get index from last operand to the getelementptr instruction
5
6   get virtual function table for class_type
7   add 2 to index to get real index into virtual table
8   get virtual_function from virtual table using index
9   add dependency (call_site.caller, virtual_function) to depgraph
10
11  // Now repeat for all derived types
12  for each type in class_type.derived_types:
13    get virtual function table for type
```

```
14      add 2 to index to get real index into virtual table
15      get virtual_function from virtual table using index
16      add dependency (call_site.caller, virtual_function) to depgraph
```

Note that it is not enough to simply retrieve the virtual function for the class itself. We also need to retrieve all derived implementations of the virtual function because they are also candidates that could be called. Thus, we repeat the process of retrieving the virtual functions for all derived classes and place dependencies on all of them.

3.4.2.4 Test Registration

To be able to run only a subset of tests, we must have the knowledge of the *exact* names of all tests in the module, including the test case name, the test name, and any prefixes or suffixes generated by Google Test. By default, the LLVM Pass only knows of the test functions that exist as C/C++ functions. The Google Test `TEST(TestCase, TestName)` macros map tests to functions that are named `<TestCase>_<TestName>_Test::TestBody()` (see Listing 2.2). Thus, we have enough knowledge to reconstruct the test case name and test name. However, there are special types of tests in Google Test called *parameterized* tests that cause Google Test to generate a prefix or a suffix to the test name when running the test. We do not have enough information from within the LLVM Pass to retrieve these prefixes or suffixes.

The solution we implement to solve this dilemma and retrieve all tests that are in the test module is to first run the test executable with the flag `--gtest_list_tests` during the initialization phase of the pass. This will give

us the output of all tests that exist in the module including their exact names. We parse this output and save the test case name, test name, prefix, and suffix of each test. We call this process *registering* the tests in the Pass. Then, as we encounter functions in the Pass that contain the substring `TestBody()`, we can compare the names of the encountered functions to the names of the tests we initially registered. If they have the same test case name and test name, then we know which test that the function refers to, and from the saved information for the test we also know what the prefix and suffix of the test is. Now, we have enough information to reconstruct the exact name of the test that will work with the Google Test filter flag.

3.4.2.5 Finalization

Once all function calls have been analyzed, the LLVM Pass computes the modified functions for the function hashes by comparing the function hashes for the current revision to that of the previous revision. Recall that *modified* refers to functions that either did not exist before and are new or did exist but have a different hash value. After finding all modified functions, the Pass then computes all transitive dependencies by traversing the old dependency graph and new dependency graph. This is equivalent to performing a breadth-first-search (BFS) from every node that corresponds to a modified function. We add all of the functions encountered during the graph traversal to the set of modified functions. Once we have finished traversing the dependency graphs, we filter the modified functions by whether or not it is a test

function. This step is specific to the test framework being used (in our case we are using Google Test). At the end of the finalization, we now have the set of modified tests.

In this chapter, we described the design and implementation of Ekstazi++. In the following chapter, we detail the process of integrating Ekstazi++ with Google Test.

Chapter 4

Google Test Integration

Google Test [10] is one of the most widely used unit testing frameworks for C/C++. Ekstazi++ supports the latest version (at the time of the writing) of Google Test, which is 1.8.0. To integrate a project with Google Test, it is recommended to use the provided build system files distributed with Google Test. In this section, we detail the process of selecting tests with Google Test and how to integrate Ekstazi++ with the Google Test framework.

Google Test tests are defined using `TEST()` macros. The test result is based on assertions defined inside the test, which are also macros and take the form `ASSERT_*()` and `EXPECT_*()`. The tests are described using a test case and a test name. An example is shown in Listing 4.1.

Listing 4.1: Google Test test macro.

```
1 TEST(TestCaseName, TestName) {
2   EXPECT_EQ(1, 1);
3 }
```

The full test name of the example test is `TestCaseName_TestName`, and it defines a non-fatal assertion that 1 should equal 1. In the following sections, we describe the different types of tests that Google Test supports as well as describe how those tests appear in bitcode and how they can be selected.

4.1 Test Filtering

To retrieve all tests that are defined in an executable, we run the executable with the flag `--gtest_list_tests`. This will print all test cases and tests to the standard output. Then, to select the tests, we run the executable using another flag, `--gtest_filter`. This flag accepts inputs that are in the same format as the output of running the executable with the `--gtest_list_tests` flag. For example, given the test suite in Listing 4.2, the result of running the executable with the `--gtest_list_tests` flag would be as shown in Listing 4.3.

Listing 4.2: Google Test example test suite.

```
1 TEST(TestCase1, Test1) {
2   EXPECT_EQ(1, 1);
3 }
4
5 TEST(TestCase1, Test2) {
6   EXPECT_EQ(1, 1);
7 }
8
9 TEST(TestCase2, Test1) {
10  EXPECT_EQ(1, 1);
11 }
```

Listing 4.3: Output of running `./test --gtest_list_tests`.

```
1 Running main() from gtest_main.cc
2 TestCase1.
3   Test1
4   Test2
5 TestCase2.
6   Test1
```

All of the test cases and test names are listed. From now on, we will refer to the representation of a test when output from the `--gtest_list_tests`

flag as the *filter representation*, since this representation is used during test selection with the `--gtest_filter` flag. The flag accepts a list of patterns that are separated with ":". An example is given in Listing 4.4.

Listing 4.4: Google Test filter flag format.

```
1 ./test --gtest_filter=<PATTERN1>:<PATTERN2>:...
```

The format of these patterns are `TestCaseName.TestName`. An example is given in Listing 4.5.

Listing 4.5: Example Google Test filter that only runs `TestCase1.Test1`.

```
1 ./test --gtest_filter=TestCase1.Test1
```

In this example, we would select the single test with the test case name `TestCase1` and the test name `Test1`. The patterns are allowed to contain a "*" character that matches any string. Thus, to select all tests from the test case `TestCase1`, we would use the pattern shown in Listing 4.6.

Listing 4.6: Google Test filter that runs all tests in the test case `TestCase1`.

```
1 ./test --gtest_filter=TestCase1.*
```

This glob character is useful for selecting parameterized tests, which are described in the following sections.

The algorithm for test selection can be summarized as:

1. Run the executable with the flag `--gtest_list_tests` and parse the output to retrieve all tests and their filter representations.

2. During finalization of the Ekstazi++ LLVM Pass, find which of the modified functions are tests.

3. Match the tests' LLVM IR representations to their previously retrieved filter representations.

4. Run the executable with the flag:
   `--gtest_filter=ModifiedTest1:ModifiedTest2:....`

4.2 Test Types

The test types for Google Test include: (1) *normal tests*, (2) *Typed Tests*, (3) *Value-Parameterized Tests*, and (4) *Type-Parameterized Tests*. Each of these tests appear differently in both the filter format and in bitcode. In the following sections, we describe the different types of tests as well as the way we match the LLVM IR to the filter representation.

4.2.1 Normal Tests and Fixture Tests

Normal tests and Fixture Tests both appear identical in LLVM IR. Normal tests are tests that simply use the `TEST()` macro. An example of a normal test is in Listing 4.7.

Listing 4.7: Normal test source code.

```
1 TEST(TestCase, TestName) {
2   EXPECT_EQ(1, 1);
3 }
```

Fixture Tests are tests that reuse the same data. First, we define a fixture class that extends the Google Test `::testing::Test` class, and then we write tests using the `TEST_F` macro. Fixture classes can define a `SetUp()` and `TearDown()` method. An example of a Fixture Test is in Listing 4.8.

Listing 4.8: Fixture Test source code.

```
1 class FixtureTest : public ::testing::Test {
2   protected:
3     int data;
4
5     virtual void SetUp() {
6       data = 5;
7     }
8
9     virtual void TearDown() {}
10 };
11
12 TEST_F(FixtureTest, TestName) {
13   EXPECT_EQ(5, data); // we have access to 'data' here
14 }
```

Note that the first argument to `TEST_F` is the name of the class, which is `FixtureTest`.

Normal tests and Fixture Tests differ in their representation in source code, but look identical in the filter representation and in LLVM IR.

Running the test executable with `--gtest_list_tests` yields the output that is shown in Listings 4.9 and 4.10 for the normal test and Fixture Test, respectively.

Listing 4.9: Normal test filter representation.

```
1 TestCase.
2   TestName
```

And for the Fixture Test:

Listing 4.10: Fixture Test filter representation.

```
1 FixtureTest.
2   TestName
```

When compiled to LLVM IR, the relevant bitcode function names for the test bodies are shown in Listings 4.11 and 4.12 for the normal test and Fixture Test, respectively.

Listing 4.11: Normal test LLVM IR representation.

```
1 TestCase_TestName_Test::TestBody()
```

And for the Fixture Test:

Listing 4.12: Fixture Test LLVM IR representation.

```
1 FixtureTest_TestName_Test::TestBody()
```

The normal test and Fixture Test both follow the same LLVM IR representation, which is the test case name followed by the test name. It is important to note that the actual `TEST()` macro produces much more bitcode than just the above function, but the actual function that contains the body of the test always contains the string `TestBody()`. In this case, it is easy to parse the IR representation to retrieve the test case name and the test name. Once we know the test case name and test name, we know the test's filter representation as well.

4.2.2 Typed Tests

Typed Tests allow clients to test multiple implementations of a common interface. The test logic is repeated over a set of types, which makes Typed Tests useful for testing polymorphic classes. A simple example of a Typed Test can be found below in Listing 4.13.

Listing 4.13: Typed Test source code.

```
1 class A {
2 protected:
3     int f;
4
5 public:
6     A(int f) {
7         this->f = f;
8     }
9
10    virtual int getF() {
11        return f;
12    }
13 };
14
15 class B : public A {
16 public:
17    B(int f) : A(f) {}
18
19    virtual int getF() {
20        return f + 1;
21    }
22 };
23
24 template <typename T>
25 class TypedTest : public ::testing::Test {
26 protected:
27    T* val;
28 };
29
30 typedef ::testing::Types<A, B> MyTypes;
31 TYPED_TEST_CASE(TypedTest, MyTypes);
32
33 TYPED_TEST(TypedTest, Test1) {
34    this->val = new TypeParam(3);
35    EXPECT_EQ(3, this->val->getF());
36 }
```

It is important to note that Typed Tests use the `TYPED_TEST` macro and require a definition of types before defining the actual tests (lines 30-31) in 4.13. The test bodies have access to the type `TypeParam`, which will be one

of the types defined by the user. Since this example defines two types, `A` and `B`, the Typed Test `TypedTest.Test1` will be run twice, once where `TypeParam` is `A` and once where `TypeParam` is `B`.

Assuming the source code is compiled to a single executable named `unittest`, running `./unittest --gtest_list_tests` will produce the output shown in Listing 4.14.

Listing 4.14: Typed Test filter representation.

```
1 TypedTest/0. # TypeParam = A
2   Test1
3 TypedTest/1. # TypeParam = B
4   Test1
```

As expected, there are actually two versions of the test case `TypedTest`, which Google Test names `TypedTest/0` and `TypedTest/1`. By analyzing the output, we can see that the parameter index `0` corresponds to the parameter type `A` and the parameter index `1` corresponds to the parameter type `B`. The LLVM IR output of the Typed Test can be found in Listing 4.15.

Listing 4.15: Typed Test in LLVM IR.

```
1 TypedTest_Test1_Test<A>::TestBody()
2 TypedTest_Test1_Test<B>::TestBody()
```

There are two test body functions, which are parameterized by one of the user-defined types. To match the LLVM IR back to the filter representation, we look for the presence of the angled brackets, `<>`, immediately before the `TestBody()` string. Since we already found that `A` is the parameter index `0` from the filter representation, we know that `TypedTest_Test1_Test<A>-::TestBody()` maps to the test `TypedTest/0.Test1`. The same applies for

`TypedTest/1.Test1`.

4.2.3 Type-Parameterized Tests

Type-Parameterized Tests are similar to Typed Tests in that clients define a list of types and common test logic, but they give more flexibility by allowing clients to first define the test logic and then instantiate the test with a set of types in another place in the code. An example of a Type-Parameterized Test can be found in Listing 4.16.

Listing 4.16: Type-Parameterized Test source code.

```
1 class A {
2 protected:
3     int f;
4
5 public:
6     A(int f) {
7         this-> f = f;
8     }
9
10    virtual int getF() {
11        return f;
12    }
13 };
14
15 class B : public A {
16 public:
17    B(int f) : A(f) {}
18
19    virtual int getF() {
20        return f + 1;
21    }
22 };
23
24 template <typename T>
25 class TypeParamTest : public ::testing::Test {
26 public:
27
```

```
28 protected:
29     T* val;
30 };
31
32 TYPED_TEST_CASE_P(TypeParamTest);
33
34 TYPED_TEST_P(TypeParamTest, Test1) {
35     this->val = new TypeParam(3);
36     EXPECT_EQ(3, this->val->getF());
37 }
38
39 REGISTER_TYPED_TEST_CASE_P(TypeParamTest, Test1);
40
41 // elsewhere in the code
42 typedef ::testing::Types<A, B> MyTypes;
43 INSTANTIATE_TYPED_TEST_CASE_P(Prefix, TypeParamTest, MyTypes);
```

The logic is mostly the same as that of the Typed Test example except that in this case, we define the types and instantiate the Typed Test case after defining the test logic. The first parameter to the macro `INSTANTIATE_TYPED-_TEST_CASE_P` is a user-defined name given to the instance of the test case.

After compiling the source code to the executable `unittest`, the output for running `./unittest --gtest_list_tests` is shown in Listing 4.17.

Listing 4.17: Type-Parameterized Test filter representation.

```
1 Prefix/TypeParamTest/0. # TypeParam = A
2   Test1
3 Prefix/TypeParamTest/1. # TypeParam = B
4   Test1
```

The output is almost identical to that of the Typed Test with the exception of the prefix. If we define another set of types and instantiate the test case again, the prefix allows us to differentiate which instance of the test case is run. Similar to the Typed Tests, we also can map the type parameter

index, which is the suffix of the test case name, to the type parameter. In this case, the parameter index `0` corresponds to the type parameter `A`, and the index `1` corresponds to the type parameter `B`.

The LLVM IR that corresponds to the function bodies is shown below in Listing 4.18.

Listing 4.18: Type-Parameterized Test LLVM IR.

```
1 gtest_case_TypeParamTest_::Test1<A>::TestBody()
2 gtest_case_TypeParamTest_::Test1<B>::TestBody()
```

Luckily, the Type-Parameterized Tests have a different signature from the typed tests and follow the pattern `gtest_case_<TestCase>_::<TestName>-<Type>::TestBody()`, which allows us to differentiate between them. We simply need to look for functions in the bitcode that start with `gtest_case_` and end with `TestBody()`. Then, we know that the function is a Type-Parameterized Test and can extract the test case name and the test name from the function name. To map the function back to the correct test filter representation, we use the type parameter and the saved type parameter indices from the output of `--gtest_list_tests`.

4.2.4 Value-Parameterized Tests

Value-Parameterized Tests allow us to define common test logic and then instantiate the tests with different values. They are similar to Type-Parameterized Tests except that we parameterize values, not types. The code for a Value-Parameterized Test is shown below in Listing 4.19.

Listing 4.19: Value-Parameterized Test source code.

```
1 class A {
2 public:
3     A() {}
4
5     int getNumChars(const char* param) {
6         return strlen(param);
7     }
8 };
9
10 class ValueParamTest : public ::testing::TestWithParam<const char*> {
11
12 };
13
14 TEST_P(ValueParamTest, Test1) {
15     A a();
16
17     const char* param = GetParam();
18
19     EXPECT_GE(5, a.getNumChars(param));
20     EXPECT_LE(3, a.getNumChars(param));
21 }
22
23 INSTANTIATE_TEST_CASE_P(Prefix, ValueParamTest, ::testing::Values("meeny",
       "miny", "moe"));
```

We first define the test case `ValueParamTest` and then instantiate it with values `"meeny"`, `"miny"`, and `"moe"`. To retrieve the current value from within the test body, we use the `GetParam()` function. As with the Type-Parameterized Tests, we can define a custom prefix name to the instantiation of the test case with the defined set of values.

Assuming that the source code is again compiled to an executable `unittest`, we can retrieve the filter representation of the tests by running the command `./unittest --gtest_list_tests`, and the output is shown in Listing 4.20.

Listing 4.20: Value-Parameterized Test filter representation.

```
1 Prefix/ValueParamTest.
2   Test1/0 # GetParam() = "meeny"
3   Test1/1 # GetParam() = "miny"
4   Test1/2 # GetParam() = "moe"
```

From this output, we can retrieve the test case name and test names of all tests and the value parameter that they are instantiated with.

The LLVM IR of the Value-Parameterized Test is shown below in Listing 4.21.

Listing 4.21: Value-Parameterized Test LLVM IR.

```
1 ValueParamTest_Test1_Test::TestBody()
```

Unfortunately, Value-Parameterized Tests have the exact same appearance as normal tests and Fixture Tests. This makes it hard to differentiate between them. To solve this ambiguity, we must first register all Value-Parameterized Tests. Then, as we encounter functions in LLVM IR, we first eagerly look to see if a test with the pattern `<TestCase>_<TestName>_Test-::TestBody()` matches a Value-Parameterized Test. If not, then we know it is either a normal test or a Fixture Test.

4.3 Google Test Prefixes and Suffixes

The filter string that is passed to `--gtest_filter=` must match a test case and test name exactly. This is easy in the case of normal tests and fixture tests, but when we want to select a Typed Test or a parameterized test, we must know what prefix and suffix to pass to the Google Test filter.

For example, for the Typed Test `TypedTest.Test1` shown in Listing 4.14, the actual name of the test is `TypedTest/0.Test1` and `TypedTest/1.Test1`, so we must pass those exact strings to the filter flag to select both tests. Similarly, for the parameterized test shown in 4.20, to select all instances of the test `ValueParamTest.Test1`, we need to pass the correct prefix and suffix to the filter, which is `Prefix/ValueParamTest.Test1/0`, `Prefix/ValueParamTest-.Test1/1`, and `Prefix/ValueParamTest.Test1/2`.

4.3.1 Handling Type Suffixes

For Typed Tests and Type-Parameterized Tests, we can match the suffix, which is the type parameter index, to the type. For example, the Typed Test `TypedTest/0.Test1` has the parameter index `0`, which corresponds to the type `A` (shown in Listing 4.14). The corresponding LLVM IR function is `TypedTest_Test1.Test<A>::TestBody()`, and we can see that the type parameter of the function is also `A`. Thus, from both the filter representation and the LLVM IR, we can retrieve the type parameter to the test. This gives us enough information to pass the exact suffix necessary (e.g., `/0`, `/1`, `/2` etc.) to the filter string when selecting Typed Tests and Type-Parameterized Tests. Because Typed Tests have no prefix, we know the exact filter string to pass to the filter flag. However, since Type-Parameterized Tests not only have a suffix but also a prefix (e.g., `Prefix/TypeParamTest/0.Test1`), we need some method to select the test because the LLVM IR does not provide any information on the prefix of the test. The solution to selecting test cases with

prefixes is discussed below.

4.3.2 Handling Instance Prefixes

Type-Parameterized Tests and Value-Parameterized Tests both require users to specify a prefix name for the specific instantiation of the test case. In addition, the LLVM IR does not have the prefix in the name of the functions that represent the test bodies. Thus, if we detect that a test function has been modified, we will use a Google Test filter with the `*` character before the test case name to select all instances of the test case. For example, using the Value-Parameterized Test, if we want to run all test instances of `ValueParamTest.Test1`, we can use the filter `--gtest_filter=*ValueParamTest.Test1/0`. This will select all instances of `ValueParamTest.Test1` regardless of the prefix. Similarly, for the Type-Parameterized Test `TypeParamTest.Test1`, we pass the filter string `--gtest_filter=*TypeParamTest/0.Test1` to run all instances of `TypeParamTest/0.Test1`.

The following chapter describes the details of generating LLVM IR alongside test executables.

Chapter 5

Build System Integration

Ekstazi++ requires the generation of whole-program LLVM IR for each test executable. Generating the LLVM IR can be achieved by passing the necessary flags to the frontend Clang compiler, and we can then integrate this process to a build system. Currently, Ekstazi++ supports three popular build systems: AutoMake, CMake, and Make. This chapter discusses the process of generating LLVM IR using the supported build systems and the process of running the Ekstazi++ LLVM Pass using the Ekstazi++ Test Runner.

5.1 Generating Bitcode

The Ekstazi++ Pass requires LLVM bitcode files (.bc) or assembly language files (.ll). These files represent the source code that has been compiled into the LLVM intermediate representation format (IR).

5.1.1 Manual Compilation and Linking

One way to integrate Ekstazi++ into a project is to write a custom target that compiles the whole program into LLVM IR files and then links them into a single IR file. This can easily be achieved by using the commands:

```
1 clang++ -S -emit-llvm src.cc -o src.ll
```

```
clang++ -S -emit-llvm test.cc -o test.ll
llvm-link src.ll test.ll ... -o main.ll
llc main.ll -o unittest
```

The intermediate LLVM IR files in this case will have the `.ll` extension, which is human-readable bitcode. This method is best for integrating Ekstazi++ with a new project, as these commands can be included in custom build steps for any build system.

5.1.2 Link Time Optimization

The steps described in the section above are recommended for integrating Ekstazi++ into a new project. However, for projects with existing build system scripts, Ekstazi++ can be easily integrated without changing the scripts by using the GNU Gold Linker [9], 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.

The following steps outline this process:

```
clang++ -flto -c src.cc -o src.o
clang++ -flto -c test.cc -o test.o
clang++ -flto -fuse-ld=ld.gold -Wl,-plugin-opt=save-temps src.o test.o -o
    test_exec
```

5.2 Make

The following template can be used for most Makefile projects:

```
CC=clang
CXX=clang++
CFLAGS=-flto
CXXFLAGS=-flto
LDFLAGS="-flto -fuse-ld=ld.gold -Wl,-plugin-opt=save-temps"

make TARGET
```

5.3 AutoMake

For AutoMake projects, we pass the same flags to the `configure` tool when configuring the project.

```
CC=clang \
CXX=clang++ \
CFLAGS = -flto \
CXXFLAGS = -flto \
LDFLAGS = -flto -fuse-ld=ld.gold -Wl,-plugin-opt=save-temps \
./configure
```

5.4 CMake

CMake is a cross-platform build system that generates build files for specific platforms such as GNU Make and MSVC. To forward the compiler flags, we set the CMakeFiles variables as follows:

```

set(CMAKE_C_COMPILER clang CACHE STRING "" FORCE)
set(CMAKE_CXX_COMPILER clang++ CACHE STRING "" FORCE)

set(CMAKE_AR llvm-ar CACHE STRING "" FORCE)
set(CMAKE_RANLIB llvm-ranlib CACHE STRING "" FORCE)

set(CMAKE_C_FLAGS "-flto" CACHE STRING "" FORCE)
set(CMAKE_CXX_FLAGS "-flto" CACHE STRING "" FORCE)

```

```
11 set(CMAKE_EXE_LINKER_FLAGS "-flto -fuse-ld=gold -Wl,-plugin-opt=
       save-temps" CACHE STRING "" FORCE)
12 set(CMAKE_SHARED_LINKER_FLAGS "-flto -fuse-ld=gold -Wl,-plugin-opt
       =save-temps" CACHE STRING "" FORCE)
13 set(CMAKE_MODULE_LINKER_FLAGS "-flto -fuse-ld=gold -Wl,-plugin-opt
       =save-temps" CACHE STRING "" FORCE)
14 set(BUILD_SHARED_LIBS OFF CACHE BOOL "" FORCE)
15
16 set(GTEST_ROOT /usr/src/gtest/cmake CACHE FILEPATH "" FORCE)
```

The CMake code above may be included in the project when developing; however, it can also be used to set the CMakeCache of an existing project.

5.5 Running the Ekstazi++ LLVM Pass

After generating the LLVM IR for each test executable, running the Ekstazi++ LLVM Pass can be achieved by running the command `opt -load libekstazi-pass.so -ekstazi <BC_FILE>`. This will generate the metadata for the current revision of the code and produce a list of test filters to use for test selection.

To simplify the process of running the Ekstazi++ LLVM Pass and selecting the tests to run, we developed a Python script to automate this process. We call this script the Ekstazi++ Test Runner. The primary steps of Ekstazi++ Test Runner are:

1. Compile the source code using the specified build system (Make, CMake, or AutoMake) into test executables and LLVM IR files.

2. Run the Ekstazi++ LLVM Pass for each LLVM IR file.

3. Parse the test filters from the modified test list produced by the Ekstazi++ LLVM Pass.

4. Run the test executables with the test filters.

In the next chapter, we describe the results of evaluating Ekstazi++ with several projects.

Chapter 6

Evaluation

To evaluate Ekstazi++, we collected 11 open-source projects from GitHub. We selected these projects by filtering GitHub projects to those that were written in C/C++ and used one of the supported build systems (CMake, AutoMake, Make). The resulting projects are diverse in size and number of tests.

6.1 Subjects

This section describes subjects used in our study. Table 6.1 shows, for each subject, its name, URL, number of buildable revisions (out of latest 50), latest available SHA at the time of our study, number of lines of code (LOC) measured using the `cloc` tool, build system used by the project, and number of tests.

Abseil is a collection of C++ code that extends the C++ standard library. Abseil provides useful code not in the C++ standard library and also provides alternatives to existing C++ standard library code. It is important to note that Abseil migrated to CMake recently, so only a limited number of revisions were available.

Boringssl is Google's fork of OpenSSL. Boringssl is not intended to be

Table 6.1: Projects used in evaluation.

Project	URL	# Revisions	SHA	LOC	BuildSys	# Tests
Abseil	https://github.com/abseil/abseil-cpp	14	99477fa	53,468	CMake	1,084
Boringssl	https://github.com/google/boringssl.git	50	9b2c6a9	193,262	CMake	839
gRPC	https://github.com/grpc/grpc.git	50	17f682d	1,406,407	CMake	1,682
Kokkos	https://github.com/kokkos/kokkos.git	50	d3a9419	128,452	Make	227
Libcouchbase	https://github.com/couchbase/libcouchbase.git	50	b028b9e	98,931	CMake	303
Libtins	https://github.com/mfontanini/libtins.git	50	b18c2ce	92,707	CMake	797
OpenCV	https://github.com/opencv/opencv.git	33	9a8a964	1,018,274	CMake	21,106
Protobuf	https://github.com/google/protobuf.git	50	264e615	317,477	AutoMake	2,086
Rapidjson	https://github.com/Tencent/rapidjson.git	34	af223d4	82,536	CMake	425
Rocksdb	https://github.com/facebook/rocksdb.git	31	2c2f388	238,995	Make	2,571
Tiny-dnn	https://github.com/tiny-dnn/tiny-dnn.git	45	1c52594	181,407	CMake	294
Total	N/A	407	N/A	3,811,916	N/A	31,414
Average	N/A	37	N/A	346,538	N/A	2,856

for general purpose use and its development arose from the gradual changes that Google maintained when using OpenSSL. Boringssl uses the CMake build system.

gRPC is a modern open-source remote procedure call (RPC) framework developed by Google that is scalable and highly performant. gRPC uses HTTP/2 and Protocol Buffers to for communication between the client and server. It serves as an alternative to the popular REST+JSON combination that is used ubiquitously today. gRPC uses the CMake build system.

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. Kokkos aims to provide a common programming model for parallel computation and data management. Kokkos uses the standard GNU Make tool and provides custom bash scripts to generate the appropriate Makefile.

Libcouchbase is the C client library for Couchbase. The Couchbase Server is an open-source NoSQL database that is optimized to provide low-

latency data management for a variety of use cases such as web, mobile, and IOT applications. Libcouchbase uses the CMake build system.

Libtins is a high-level C++ library for sending, receiving, and manipulating network packets. Libtins was designed with efficiency and portability in mind. Libtins uses the CMake build system.

OpenCV stands for Open Source Computer Vision Library and is a machine learning software library that specializes in computer vision applications. OpenCV has many language interfaces such as C++, Python, Java, and MATLAB. OpenCV uses the CMake build system.

Protobuf is a language and platform agnostic method for serializing data developed by Google. Protobuf is designed for both simplicity and performance and is used for remote procedural call (RPC) systems such as gRPC. Protobuf uses the GNU AutoMake build system.

Rapidjson is a C++ library for parsing and generating Javascript Object Notation (JSON). Much of Rapidjson's design was inspired by RapidXML, a library for XML parsing. Rapidjson uses the CMake build system.

Rocksdb is an embedded database for key-value storage developed by Facebook. Rocksdb is optimized for speed and low-latency applications and is based on the log-structured merge-tree (LSM tree) data structure. Rocksdb uses the GNU Make build system.

Tiny-dnn is a header-only deep learning library written in C++. Tiny-dnn is well suited for use on computational systems with limited resources such

as embedded systems and IoT devices. Tiny-dnn uses the CMake build system.

6.2 Experiment Setup

The experiments were run on an 8-core 3.9 GHz AMD 1800X CPU with 16GB 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.

For each project, we attempted to examine the 50 most recent consecutive revisions. For each revision, we built the project using the supported build system and then ran Ekstazi++ to select and run the tests. The experimental procedure can be found in Listing 6.1. Our experiment setup closely follows recent evaluations of RTS techniques (for Java) [8, 15, 31, 34].

Listing 6.1: Experimental procedure.

```
1 function run_project(project, revisions):
2     clone project.url
3     for revision in revisions:
4         checkout revision
5         configure project
6         build project
7         select and run tests
```

6.3 Results

For each project, we compared the results of using Ekstazi++ to those of not using Ekstazi++ (i.e., retest-all). The data we collected includes the build time, number of tests selected, and amount of time spent running tests.

Time for Ekstazi++ includes all steps: analysis, execution, and collection [8]. Table 6.2 shows the total number of tests and the time to run tests for retest-all and Ekstazi++ as well as the ratio of both number of tests and time spent running tests for Ekstazi++ to those of retest-all.

Additionally, for each project we plotted the number of tests run, time to run tests, and the cumulative time; the results for each project are shown graphically below in Figures 6.1-6.11.

Table 6.2: Test selection results using Ekstazi++.

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

Project	Tests		Time (s)		Ratio	
	retest-all	Ekstazi++	retest-all	Ekstazi++	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	3083.39
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	N/A	N/A
Average	100,505	21,421	13,885	5,690	11.14	38.70*

*The average ratio for time excludes the projects Libtins and Protobuf.

In general, most projects saw a favorable reduction in both number of tests run for each revision and the time spent running tests. The average

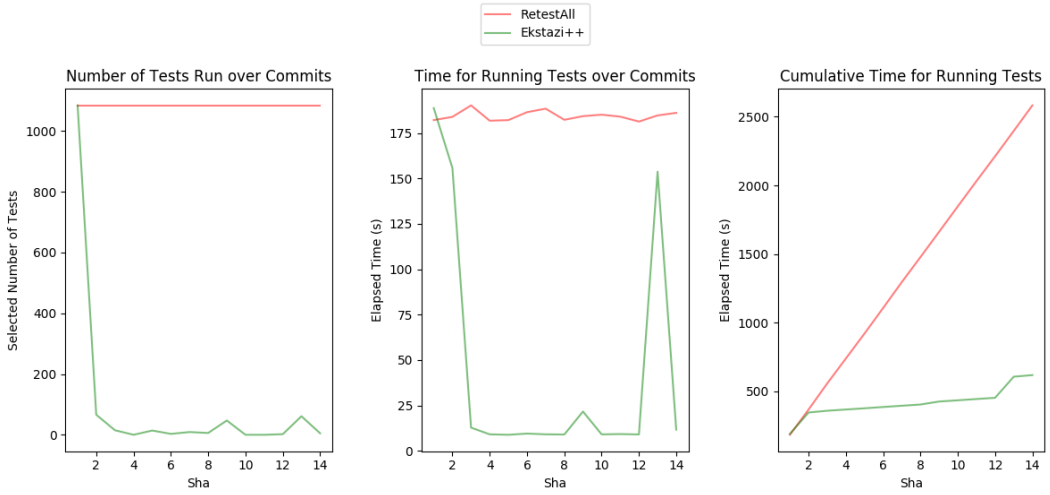


Figure 6.1: Results for Abseil.

reduction in the number of tests (if we exclude two projects with the shortest testing time) was 88.86% and the average reduction in the time taken to run tests was 61.30% with the largest reduction being 97.20% and 88.09% for the number of tests and the time to run tests, respectively. The two projects that did not exhibit a reduction in the time to run tests were Libtins and Protobuf, both of which had a much shorter amount of time (less than 10 seconds) to run tests relative to the other projects. For these projects, the time to statically analyze the bitcode and compute the new test filters took longer than the actual time to run the tests.

As mentioned earlier, Abseil migrated to the CMake build system recently, so we could only retrieve results for 14 revisions. For those 14 revisions, the results were similar to those of most of the other projects.

Kokkos exhibits a small drop in the number of tests run at the 5th

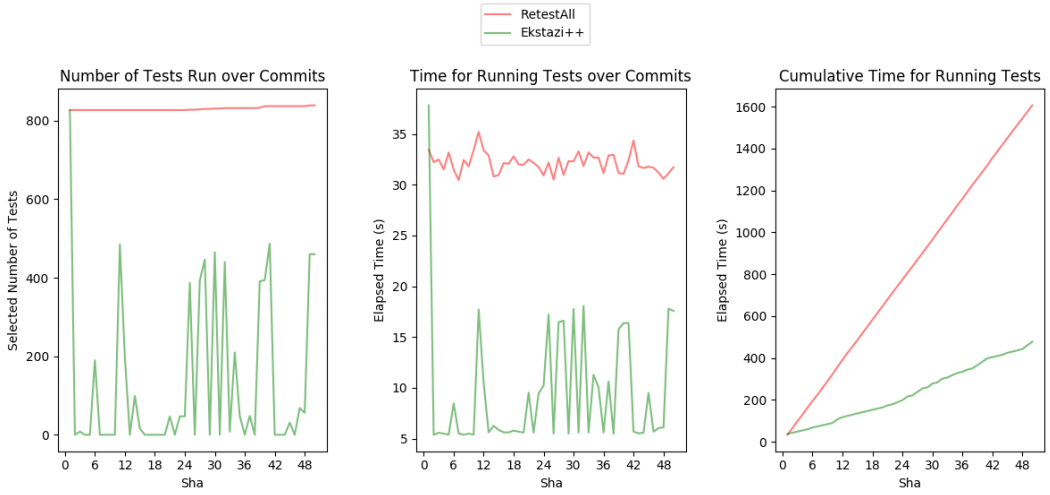


Figure 6.2: Results for Boringssl.

revision due to a failure to build one of the test executables, so fewer tests were run.

OpenCV shows a large increase in the time to run tests after the 16th commit and then a similarly large decrease in the time to run tests after the 28th commit. Manual inspection showed that the increase in the time to run tests was due to the addition of 11 costly tests that included video data, and the decrease in the time to run tests was due to the refactoring of one of the test source files that included the removal of 10 tests related to testing convolutional neural network layers.

Interestingly, the amount of time for running the tests for Tiny-dnn increases between the revisions 20-25 to over 2,500 seconds but drops back to about 1,000 seconds immediately after, while the number of tests run increased by a small amount and then stayed the same. Investigating the changes into the

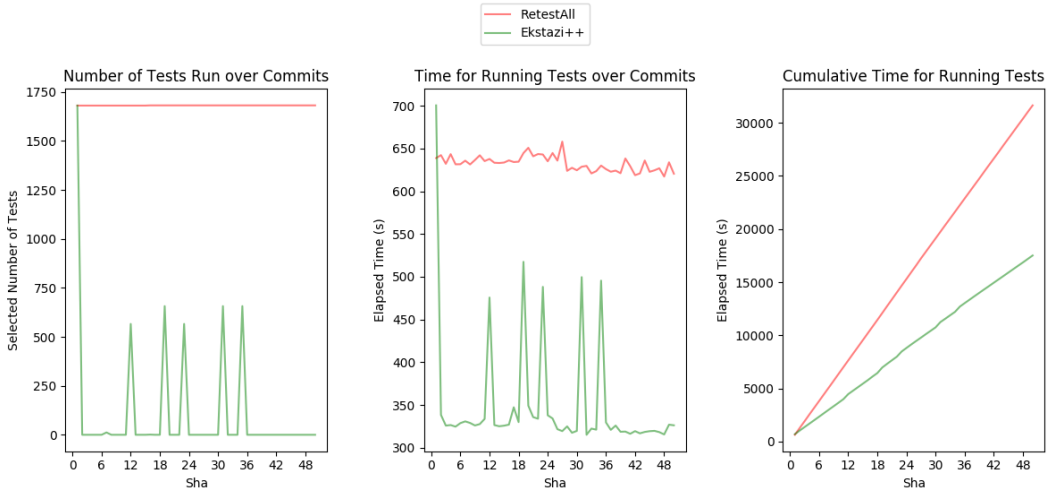


Figure 6.3: Results for gRPC.

project, we found that the modification made at revision number 19 introduced new code and tests to support recurrent neural networks. These added tests caused a large increase in the time to run tests. Revision number 24 then decreased the size of the neural network being tested, causing the time for testing to drop back to what it was before.

6.3.1 Build Time

We also compared the time it took to build each project to the time it took to run all tests for the project. The results are shown in Figure 6.12.

6.3.2 Type Hierarchy Analysis

To observe the characteristics of the type hierarchy and its potential impact on the speed of the static analysis, we recorded the number of classes,

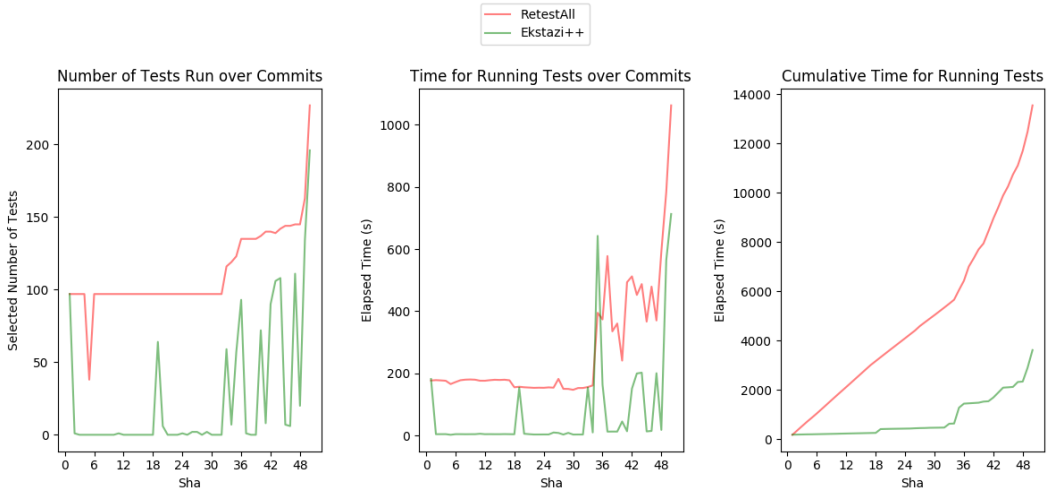


Figure 6.4: Results for Kokkos.

the maximum depth of the type hierarchy, and the average depth of the type hierarchy for each project. The results are shown in Table 6.3.

Visually, there is not an evident correlation between the depth of the type hierarchy and the time to run tests. Furthermore, it is hard to define a metric for measuring the time spent in static analysis that is caused directly by a change to the type hierarchy and not by some other characteristic of the code. For future work, we would like to further explore the effects of the type hierarchy on the Ekstazi++ LLVM Pass.

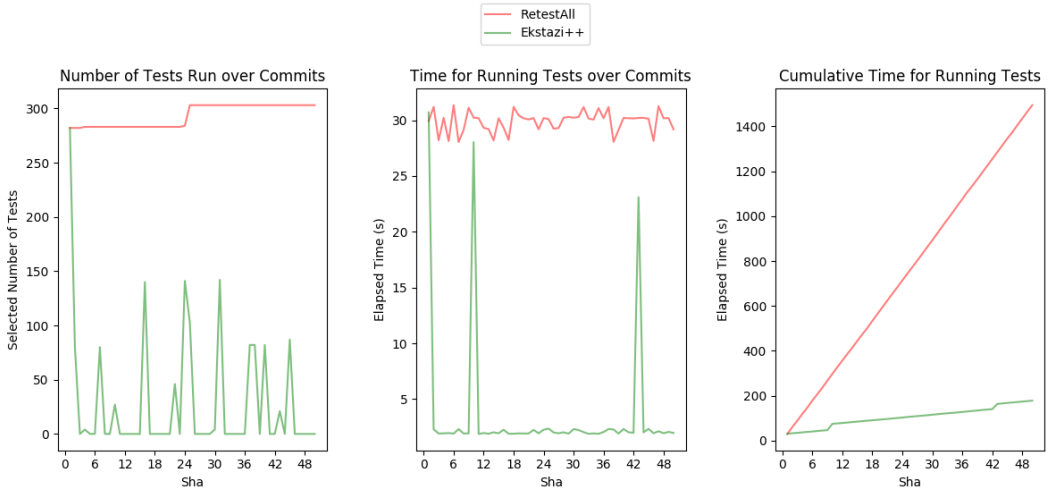


Figure 6.5: Results for Libcouchbase.

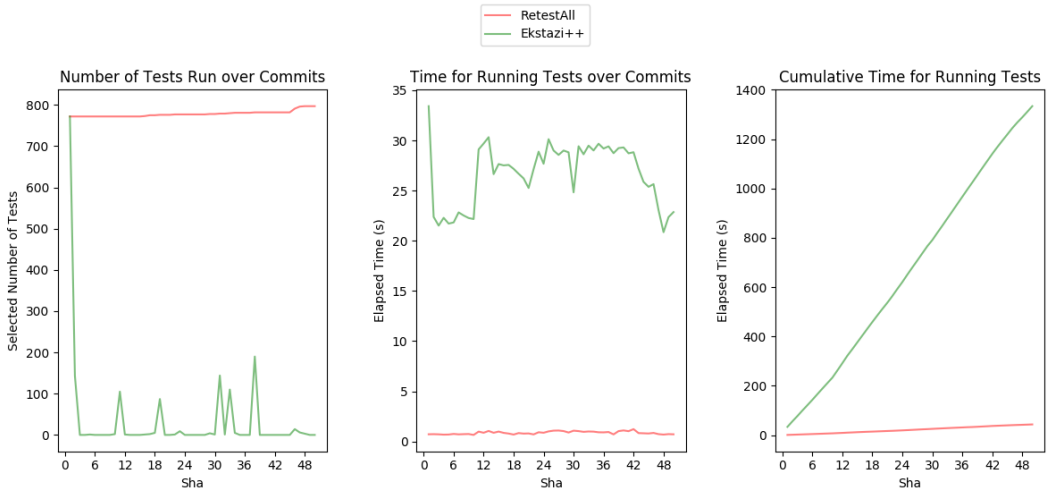


Figure 6.6: Results for Libtins.

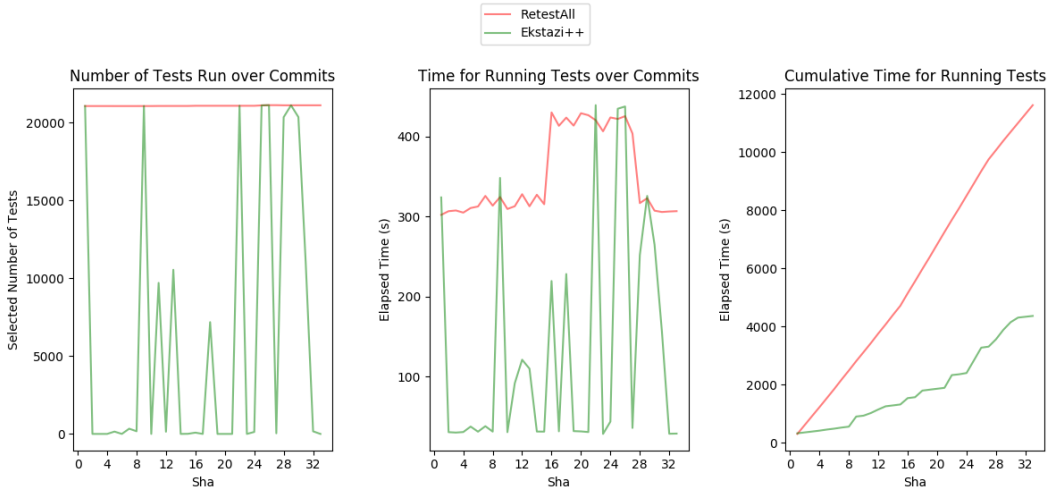


Figure 6.7: Results for OpenCV.

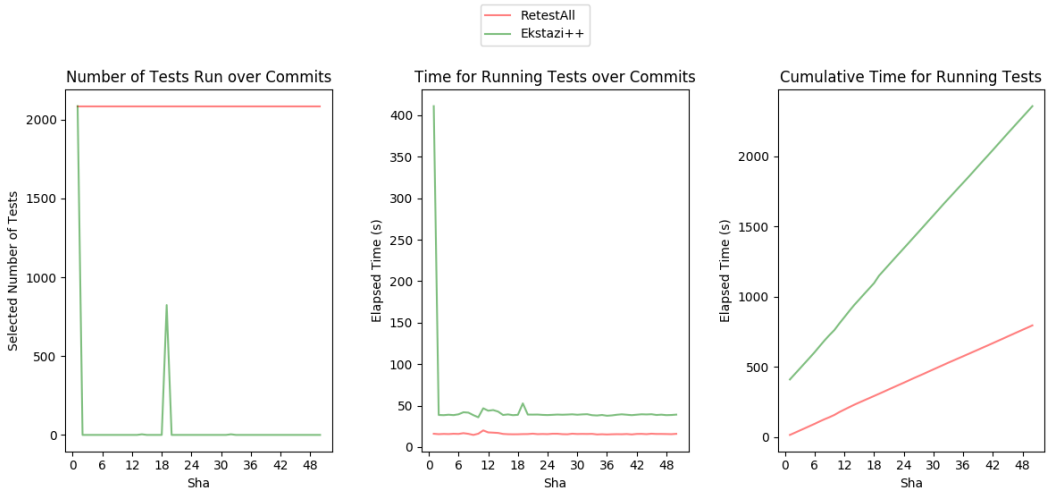


Figure 6.8: Results for Protobuf.

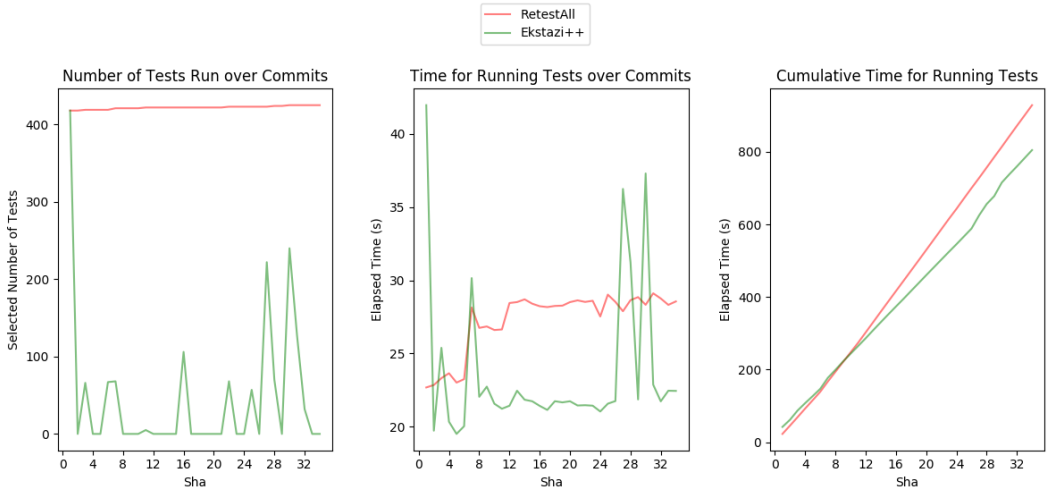


Figure 6.9: Results for Rapidjson.

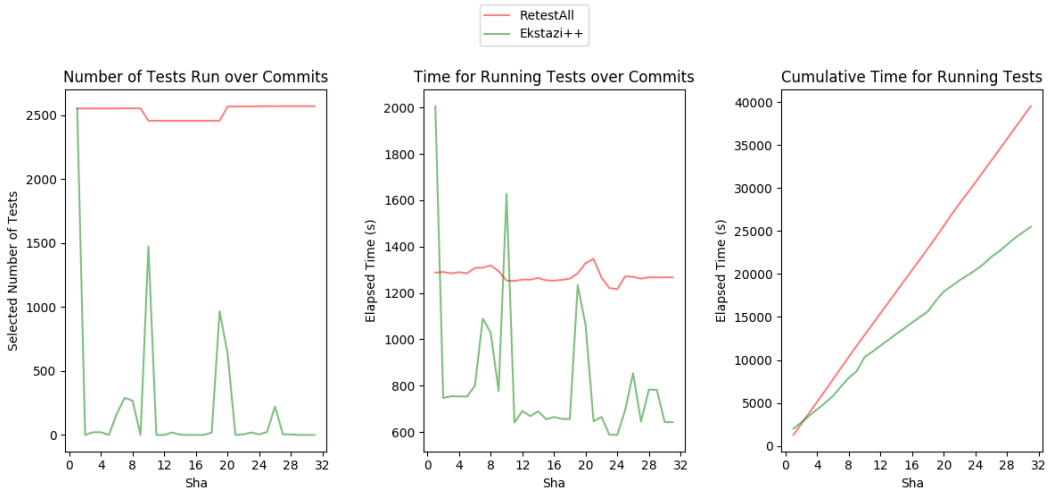


Figure 6.10: Results for Rocksdb.

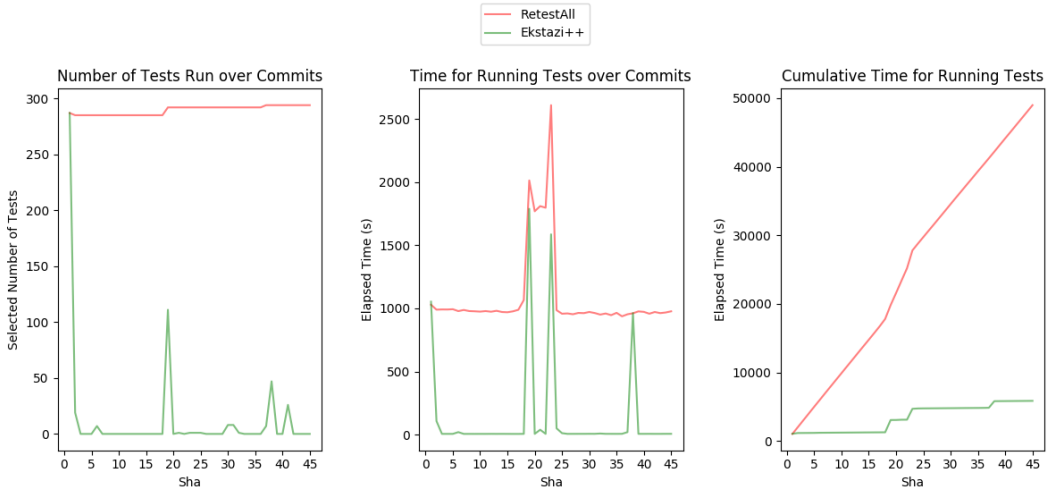


Figure 6.11: Results for Tiny-dnn.

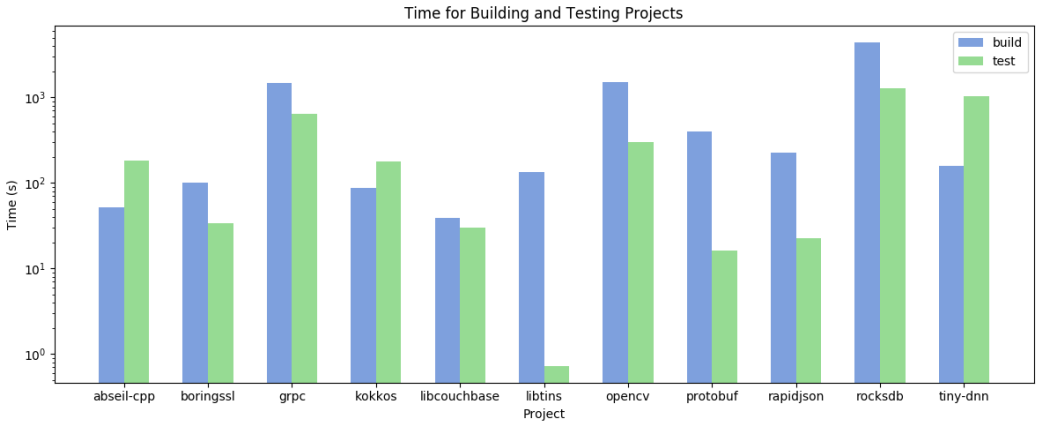


Figure 6.12: Times for build and test phases for each project.

Table 6.3: Type hierarchy information for projects.

The number of classes is the total number of classes summed over all test executables for the project. The number of derived classes is the number of classes that are derived from at least one other class. Max depth refers to the maximum depth of the class hierarchy, and average depth refers to the average depth from the roots of the hierarchy to each of their leaves.

Project	# Classes	# Derived Classes	Max Depth	Average Depth
Abseil	2,823	1,523	2	1.16
Boringssl	714	337	2	1.07
gRPC	12,621	5,485	3	1.20
Kokkos	1,386	658	2	1.09
Libcouchbase	399	39	2	1.00
Libtins	5,151	2,178	3	1.59
OpenCV	7,982	3,939	3	1.49
Protobuf	5,312	2,554	4	1.66
Rapidjson	950	465	1	1.00
Rocksdb	24,459	12,524	3	1.10
Tiny-dnn	1,236	893	2	1.56

Chapter 7

Discussion

This chapter discusses several design decisions related to Ekstazi++, current limitations, and future directions.

7.1 Safety

We manually inspected the projects while running Ekstazi++ to ensure that modifications to the code would result in tests being selected and run. However, it is cumbersome to ensure that *exactly* the correct tests are selected at each iteration without manually walking through the dependency graph, which is impractical for the size of the projects we used. We developed a suite of tests to ensure that Ekstazi++ works as intended for different types of code modifications, but it is almost impossible to test every different scenario with a language that has many complex features such as C++. Thus, Ekstazi++ is not guaranteed to be safe outside of the projects we inspected and the test cases we ran.

7.2 Limitations

We describe several limitations of Ekstazi++, which are also interesting topics to be explored in future work.

7.2.1 Dependence on Front End Compiler

Ekstazi++ currently supports static analysis of LLVM IR with Clang as the front end compiler. For C/C++ specific features such as the layout of virtual tables and mangling function names, Ekstazi++ depends on the C++ ABI specification that Clang uses, which is the Itanium C++ ABI. Both GCC and Clang implement the Itanium C++ ABI, but some other compilers may not. When we wish to extend Ekstazi++ to support other languages that use different front end compilers, we will need to define a way to support those language-specific features.

7.2.2 LLVM Pass Integration

Currently, the Ekstazi++ LLVM Pass is built as a shared library that is loaded into the LLVM `opt` tool. This is cumbersome because users need to call `opt -load <PATH_TO_EKSTAZI_LIB> libekstazi-pass.so -ekstazi <BC_FILE>` for every bitcode file that is generated. We currently have a Python script that automates this process, but in the future we hope that the Ekstazi++ LLVM Pass can be integrated into the LLVM build tree so that users can run the Pass without specifying the location of the built library (e.g. `opt -load -ekstazi <BC_FILE>`), or even more simply by specifying an optimiza-

tion level while compiling (e.g. `clang++ -O3 ...`).

7.2.3 Shared Libraries

Ekstazi++ currently supports test executables that link to the program code using static libraries. Because the code contained in static libraries is included with the executable, the full dependency graph for each test executable module is guaranteed to contain all the information such as function definitions and type metadata. However, in the case of shared libraries, the compiled bitcode for each test executable does not contain the function definitions of the shared libraries, so there is not enough information to generate the dependency graph.

One possible solution for this is to incorporate an additional input to Ekstazi++ that specifies the shared libraries that the current test executable depends on. Then, we can merge the dependency graph of the shared libraries with the graph of the test executable.

7.2.4 Difficulties with Google Test Integration

Google Test provides no programmer API, so the test selection has to be done by first listing all tests using `--gtest_list_tests` and saving the *exact* names of the tests including the test case name, test name, prefix, and suffix. We then must match functions that we encounter during the Ekstazi++ LLVM Pass to the saved test names in order to generate the correct test filters to select the tests. This process is cumbersome as it adds an unnecessary dependency

on the test executable itself. It could be more effective to integrate Ekstazi++ with a test framework that defines a programmatic API that we can use to get the exact names of the tests and then select the modified tests without needing to run the test executable during the Ekstazi++ LLVM Pass.

7.3 Future Work

We document several directions that would be interesting to explore in the future.

7.3.1 Optimizing Virtual Call Dependencies

The current implementation of Ekstazi++ places dependencies on a class's virtual function as well as all derived implementations of the virtual function. However, it is common that a test in reality depends only on a specific derived class's implementation of a virtual function. For example, consider the code in Listing 7.1.

Listing 7.1: Example code for a test that depends only on a derived implementation of a virtual function.

```
1 class A {
2   virtual int foo() { return 10; }
3 };
4
5 class B : A {
6   virtual int foo() { return 20; }
7 };
8
9 class C : A {
10  virtual int foo() { return 30; }
11 };
12
```

```
13 A* create_instance {
14   A* b = new B();
15   return b;
16 }
17
18 TEST(Test, B) {
19   A* a = create_instance();
20   EXPECT_EQ(20, a->foo());
21 }
```

In the current version of Ekstazi++, we would add a dependency to `A::foo()`, `B::foo()`, and `C::foo()` from `Test.B` because the type of `a` is `A*`, and both `B` and `C` are derived types of `A`. However, we really only want to place the dependencies on `A::foo()` and `B::foo()` since the class `C` is never used by the test.

One possible solution to reduce the number of dependencies on virtual functions is to keep track of constructor calls in the code and to only place a dependency on a virtual function if the test actually instantiates the class that the virtual function belongs to directly or indirectly. In the example above, we can visualize that in the dependency graph, the test `Test.B` calls the function `create_instance()`, and the function `create_instance()` calls the constructor `B::B()`. Thus, `Test.B` *indirectly* constructs the class `B`, so we know to only place the dependency on `A::foo()` and `B::foo()`.

7.3.2 Supporting Other Languages and Test Frameworks

Since Ekstazi++ integrates with the LLVM backend, it can be extended to support other languages that support compilation to LLVM IR. Language-

specific features would have to be added to Ekstazi++, such as support for inheritance or other forms of indirect calls. However, the majority of the test selection logic is already built into Ekstazi++ and allows for simple integration with other languages.

Similarly, Ekstazi++ can be extended to support other test frameworks. The current implementation supports Google Test, which is one of the most popular test frameworks for C/C++. The code in Ekstazi++ defines an interface for *test adapters*, so integrating a different test framework is as simple as implementing the test adapter interface.

7.3.3 Class-Level Dependencies

Ekstazi++ currently supports function-level dependencies. We find functions that are modified in the code and traverse the dependency graph to see which test functions are affected by the changes. In several of the open source projects we tested, the bulk of the time spent during the Ekstazi++ LLVM Pass is computing the checksum for each function and traversing the dependency graph on a function-level granularity. One solution to reducing the time spent on analysis is to instead use a class-level granularity, similar to the approach used by Ekstazi [8]. However, since C++ does not require functions to be defined inside classes as Java does, it would be interesting to observe how many of the open source projects utilize classes sufficiently to benefit from this optimization.

7.3.4 Dynamic Function Call Tracing

The bulk of the difficulty in the static analysis that Ekstazi++ uses lies in supporting language-specific features such as virtual calls and function pointers, which are simply indirect calls in LLVM IR. At compile time, it is difficult to know which function is actually being called during an indirect call because LLVM does not officially support retrieving the callee from an indirect call site. Currently, we rely on the Itanium C++ ABI specification to statically compute the callee.

A possible solution is to combine the static analysis with a dynamic analysis tool such as the LLVM XRay instrumentation tool [32]. XRay allows for dynamic tracing of function calls, which would eliminate the need to statically compute the callee from an indirect call site. We can instead build the dependency graph at runtime using XRay and use a combination of the statically-computed function hashes with the dynamically-computed graph to select tests [4].

7.3.5 Flaky Tests

Flaky tests are tests that may pass or fail for the same input if run multiple times [11, 18–20]. These tests give nondeterministic results and are usually due to a dependence on concurrency, an unstable environment, or other potentially undefined behavior. Ekstazi++, similar to incremental build systems, does not specially treat flaky tests.

Chapter 8

Prior Work

Regression test selection (RTS) has been studied for decades; several (not-so-recent) surveys nicely summarize prior work on RTS [1, 6, 33]. In this section we briefly present most closely related work and contrast the prior work with the work presented in this thesis.

Early techniques supported projects written in C/C++ [4, 14, 23, 25]. Rothermel and Harrold [23] presented a technique (for C) that builds control dependency graphs to detect affected tests; this work was evaluated in later years [24]. In their work, dependencies for tests were collected dynamically. Chen et al. [4] developed a technique that combines static and dynamic analysis for C; the dynamic part was used to capture function invocations via pointers. Kung et al. [14] introduced a class firewall approach to detect affected tests by tracking dependencies among classes. Rothermel et al. [25] 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. Ekstazi++, 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.

Ren et al. [22] 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. [13] 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. [21] 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 [24]. 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, Ekstazi++ is most closely related to Chianti and the work on dangerous edges. Unlike prior work, Ekstazi++ targets C++, does not analyze source code but binaries, and does not assume availability of any dynamically computed coverage (but computes coverage statically). Ekstazi++ is also evaluated on much larger set of projects.

Gligoric et al. [8] presented Ekstazi, an RTS technique based on dynamic class-dependencies for Java projects. This initial work on Ekstazi was among the first to evaluate an RTS implementation on many large open-source

projects using commits that are available in public repositories. Ekstazi has been adopted by both open-source projects and industry. Ekstazi# [30] implements the Ekstazi technique for .NET platform. Legunsen et al. [15] 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 [34] presented a hybrid technique that dynamically tracks dependencies on both methods and classes. Recently, Wang et al. [31] introduced the first refactoring-aware RTS, i.e., an RTS technique that does not run tests that are affected only by behavior-preserving transformations. Like these recent RTS projects, Ekstazi++ is also extensively evaluated using open-source projects and large number of revisions available in public repositories of those projects. Unlikely recent work, Ekstazi++ targets C/C++ projects that compile to LLVM bitcode and use Google Test. Additionally, Ekstazi++ tracks dependencies on a 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 [8,15,34], and our findings in this thesis show that function-level granularity can provide substantial reduction in test execution time for C++ projects. An interesting future direction is to develop a coarse-grained RTS technique for C++ by extending Ekstazi++ and compare its performance with the technique presented in this thesis.

Chapter 9

Conclusion

This document presented a novel RTS technique named Ekstazi++, which targets projects written in C++ that use the LLVM compiler and the Google Test testing framework. Ekstazi++ implements an RTS technique based on call graph analysis; to ensure correctness in the presence of inheritance, Ekstazi++ analyzes differences in call graphs obtained from two revisions. Ekstazi++ integrates with many existing build systems, including AutoMake, CMake, and Make. Ekstazi++ 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 Ekstazi++ 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 Ekstazi++ 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 thesis, Ekstazi++ can be a valuable addition to any large project that uses continuous integration system (i.e., runs tests frequently) and has tests that run for long time.

Bibliography

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] Atif M. Memon and Myra B. Cohen. Automated testing of GUI applications: Models, tools, and controlling flakiness. In *International Conference on Software Engineering*, pages 1479–1480, 2013.

[20] 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.

[21] 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.

[22] 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.

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

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

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

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

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

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

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

[30] 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.

[31] 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.

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

[33] 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.

[34] Lingming Zhang. Hybrid regression test selection. In *International Conference on Software Engineering*, 2018. To appear.

[35] 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.

Vita

Ben Fu was born in Hunan, China on 25 August 1995, the son of Dr. Minghua Fu and Wei Peng. He moved with his family to Plano, Texas in 2000 and graduated from Plano Senior High School in May of 2013. He attended The Univeristy of Texas at Austin from 2013-2017 as an undergraduate and applied and was accepted into the BSEE/MSE Integrated Program at UT Austin in 2017. He is currently studying to graduate with both a Bachelors of Science in Electrical and Computer Engineering and a Master of Science in Software Engineering.

Permanent address: 7113 Collin McKinney Pkwy
McKinney, Texas 75070

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.