THE UNIVERSITY
of EDINBURGH

Software Testing - Test Suite Compilation and Execution Optimizations

Panagiotis Stratis

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2020

Abstract

The requirements and responsibilities assumed by software have increasingly rendered it to be large and complex. Testing to ensure that software meets all its requirements and is free from failures is a difficult and time-consuming task that necessitates the use of large test suites, containing many test cases.

Time needed to compile and execute large test suites has become prohibitive. Current optimization techniques aim to reduce the test suite size by removing redundant test cases. However, as systems become larger, the number of essential test cases is still very large and affects the software life-cycle.

In this thesis, we explore techniques for reducing the compilation and the execution time of test suites without removing any test cases or changing computing infrastructure. All of our proposed techniques can be used in conjunction with existing test suite optimisations.

1. For test suite compilation, we propose a data transformation that reduces the number of instructions in the test code, which in turn reduces compilation time. Using two well known compilers, GCC and Clang, we conduct empirical evaluations using subject programs from industry standard benchmarks and an industry provided program. We evaluate *compilation speedup*, *execution time*, *scalability* and *correctness* of the proposed test code transformation.

2. For test suite execution, we propose a novel approach to improve instruction locality across test case executions. Our approach measures the distance between test case executions (number of different instructions). We then schedule the test cases for execution so that the distance between neighboring test cases is minimised. We empirically evaluate our approach with 20 subject programs and test suites from the SIR repository, EEMBC suite and LLVM Symbolizer to compare execution times and cache misses with test case orderings using our approach versus a traditional ordering maximising coverage and random permutations. We also assess overhead of algorithms in generating orderings that optimise instruction cache locality.

3. In our final contribution, we target execution time of heterogeneous test suites and assess the effect of device-based test case scheduling. We propose a test case scheduling algorithm which improves the load balancing between multiple devices of a heterogeneous system in an attempt to reduce the overall test

suite execution time. We conduct empirical evaluation on a large-scaled, industrial test suite targeting implementations of the SYCL standard which has been developed by Codeplay Software.

The outcome of our research can be summarized as follows:

1. Our data transformation approach resulted in significant compilation speedups in the range of 1.3× to 69×. Our experiments show that the gains in compilation time allow significantly more test cases to be included in test suites, improving scalability of test code compilation.

2. Our instruction-based test case scheduling algorithms were able to achieve a maximum execution speedup of 29.48%. Performance gains were considerable for programs and test suites where the average number of different instructions executed between test cases was high.

3. Finally, we found that a maximum of 25.42% speed-up is achieved by our device-based test scheduling algorithm when compared to parallel test case execution of a heterogeneous test suite without test scheduling.

Our proposed techniques are able to significantly reduce the compilation as well as the execution time of test suites without eliminating any test cases or upgrading computing infrastructure. Our data transformation results in faster test code compilation while our test case scheduling algorithms achieve significant speed-ups for programs executing on single-CPU, multi-CPU as well as heterogeneous architectures.

As systems get more complex, they require frequent and extensive testing. Our techniques provide safe and efficient means of compiling and executing test suites which, in combination with existing test suite optimisations, can significantly reduce the cost of software testing.

Lay Summary

Software testing is the process of ensuring that a given software works as expected. Software engineers are testing their software by developing scenarios in which they execute the software under specific conditions and then verify that the outcome matches their expectations. As software systems become more complex, the number of scenarios needed for testing all aspects of a system becomes very large and the time it takes for these scenarios to run makes testing infeasible.

Until now, the software community has been addressing this problem by removing scenarios that are considered obsolete. However, the number of essential scenarios for testing a complex system is still very large and prohibits frequent testing. In this work, we follow a different approach and instead of removing scenarios, we reduce the time of testing by transforming these scenarios into a form that is faster to be processed by the computer as well as by running these scenarios in specific orders which efficiently utilize the underlying computer hardware. Our research complements the approaches proposed by industry and academia in an attempt to improve the feasibility of software testing.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Dr. Ajitha Rajan. She supported me in every way possible. Her endless energy and positivity helped me keep going in difficult periods. She is also very patient with unorganised people who are always late (or not showing up at all) in meetings like me. Without her, this thesis would be impossible.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Björn Franke and Prof. Michael Rovatsos for their insightful commends which helped me transforming the publications into a (hopefully) readable thesis.

I would also like to thank Codeplay Software for accepting me in their family and allowing me to contribute to the SYCL standard as well as to experiment with industrial software. A special thanks to Gordon Brown - the C++ template guru and an awesome manager. I still remember the legendary lunchtime board games.

A huge thank you my fellow colleagues: Vanya Yaneva and Chao Peng. Vanya for being the organized one when analysing the results, for being there for me when I wanted somebody to talk to and for reminding me that I behave like a child when I am arguing (true story). Chao for being the coolest guy in the forum and for always being willing to help (remember that night with the *float* bug in our LLVM pass?).

Last but not least, I would like to thank my dad Dimitris, my sister Dina and my wife-to-be Martina for being in my life.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Panagiotis Stratis*)

To my Mother.
I miss her a lot.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software is an intrinsic part of our everyday lives with worldwide spending currently at 314 billion USD[1]. Testing to ensure that the software meets its requirements is a notoriously hard and time consuming process, often representing 50% of the cost of software development [67].

On a high level, the testing process, or test *workflow*, consists of four main *artifacts*: the *software under test*, which can vary from a complex application to a single function, the *software specification* that defines its intended behaviour, a set of *test inputs* each of which is passed to the *software under test* to initiate a test case execution and a *test oracle* which observes the test case execution and decides whether it is successful (i.e. the software abides by its specification) or failed. The *test oracle* itself is derived from the *software specification*. A *test case* is composed of a *test input* and the *test oracle*. Finally, a collection of test cases is defined as a *test suite* and the process of executing them as *test suite execution*. Figure 1.1 illustrates the test workflow *artifacts*.

Research in the software testing community has primarily focused on the automatic generation of *test inputs* as well as *test oracles*. For *test inputs*, the research effort has resulted in the creation and evaluation of test adequacy criteria and the development of automatic *test input* generation tools. When it comes to *test oracles*, researchers have focused on ways of automatically generating models which are able to map a *test input* to an expected output given a *software specification* artifact. An area, however, that has not been given enough attention is the process of test suite execution and, more specifically, its *efficiency* - an important aspect given the complexity of modern software and the difficulty in testing it.

[1] http://www.statista.com/statistics/203428/total-enterprise-software-revenue-forecast/

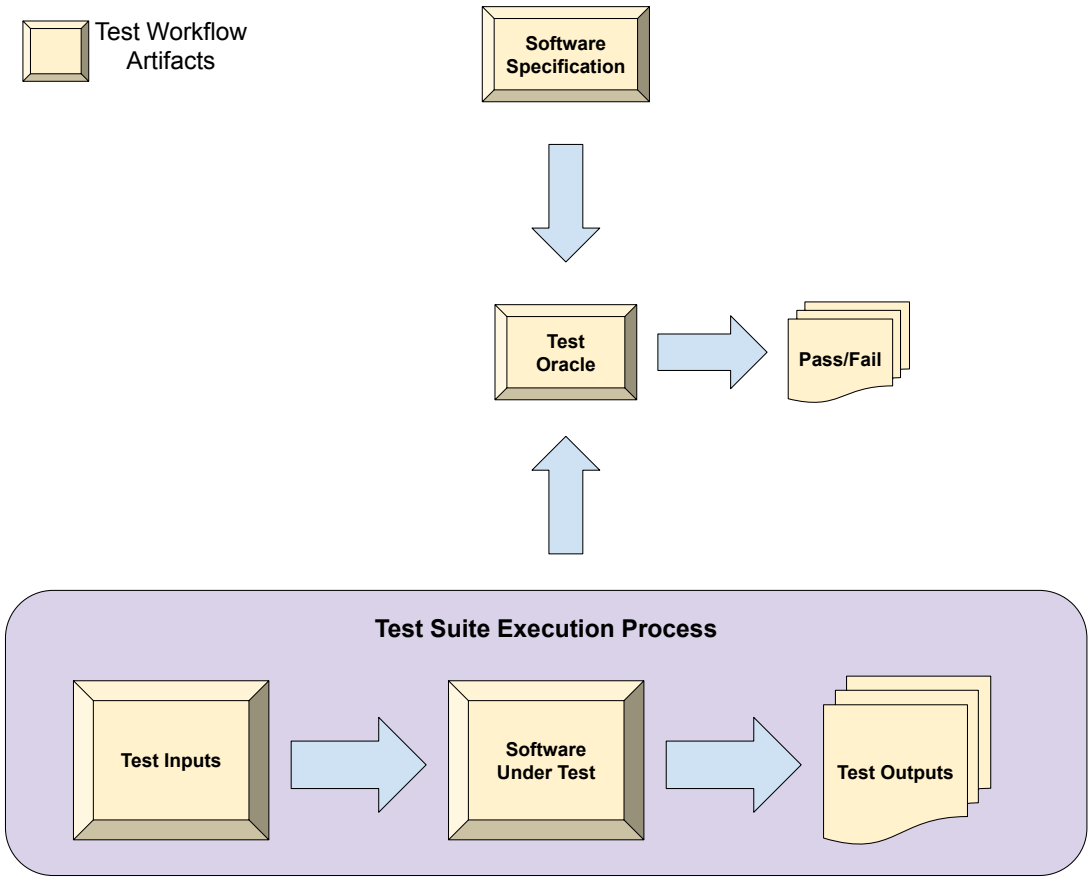

Figure 1.1: Test Workflow Artifacts

1.1 Feasibility of Testing

As the scale and complexity of software increases, the number of tests needed for effective validation becomes extremely large, slowing down development, hindering programmer productivity, and ultimately making development costly [165, 164]. The need for large numbers of tests is magnified in agile software development practices, like Continuous Integration (CI) and Test-Driven Development (TDD), that require extensive testing to be performed [11, 84, 52].

Software companies are able to confirm this observation. Google, who use CI development for their products, report a need for running more than 100 million tests per day [111]. Microsoft report that testing code changes is time consuming and annual cost of regression testing exceeds tens of millions of dollars [74]. Codeplay Software [166], who develop specialised tools, including compilers, runtimes and debuggers for heterogenous systems, use CI for their development, which necessitates frequent compilation and running of large numbers of tests, taking huge amounts of

time.

Over the last decades, the research community has focused on discovering ways for reducing test *workflow* time. Code coverage metrics are being used by industry and academia to describe the degree to which a program is tested by a test suite. The main idea behind existing optimization techniques is to achieve high coverage with as **few** test cases as possible. However, reducing the size or selecting a subset of test cases from a test suite has been shown to also reduce its fault-finding capability [70]. In addition, as systems become larger and more complex, the number of test cases needed for achieving acceptable levels of coverage is still very large [122].

1.2 The Test Suite Execution Process

Current test suite optimization techniques attempt to reduce test suite execution time by reducing the number of test cases. What are, however, the actual bottlenecks of the test suite execution process given a large number of test cases? The test suite execution process is being treated by academia and industry as an abstract *artifact* where test cases are being added or removed in order for certain criteria to be satisfied. If we want to explore alternative ways of reducing test time, rather than just remove test cases, we first need to understand the steps involved in the test suite execution process. As shown in figure 1.2, the test suite execution process is more complex than the simplistic model of just executing the *software under test* with different *test inputs* and observing its behaviour. The test suite execution process consists of three phases:

- ***Test suite compilation -*** This is the phase where a test suite is compiled into an executable program. It is a phase that exists solely for languages that include the concept of compilation in some form (e.g. C++ and JAVA).

- ***Test case scheduling -*** During this phase, the execution order of the test cases is determined. The form of the execution order depends on whether test cases can be executed in parallel.

- ***Test case execution -*** This phase entails the actual execution of the test cases which is setting up the environment, executing the *software under test* with the *test input*, observing the test output and comparing it to the expected behaviour.

1.3 Problem Statement

Increasing numbers of test cases present a challenge for all the phases of test suite execution process and make up a large fraction of the overall testing cost [218]:

- ***Test suite compilation -*** The size of the test code (i.e. the test suite) is proportional to the number of test cases. Compiling large pieces of test code can be excessively time consuming especially when the compiler is configured to apply its full set of optimizations, something which is almost always the case.
- ***Test case scheduling -*** The complexity of non-trivial test case scheduling algorithms is proportional to the number of test cases. As this number increases, test case scheduling can dramatically increase the test *workflow* time. For a test suite with *N* test cases, the asymptotic complexity of a test scheduling algorithm is at least *O(N)* in order for all test cases to be considered.
- ***Test case execution -*** Every test case execution entails at least one execution of the *software under test*. Therefore, the more test cases are to be executed, the more executions of the *software under test* will occur and the more this phase will last. The impact on the overall test *workflow* time is proportional to the number of test cases as well as to the average execution time of each test case.

1.4 Objective and Contributions

The *research objective* of this thesis is to reduce the test *workflow* time **without** eliminating any test cases or changing computing machine. To that end, this thesis focuses on the test suite execution process of test *workflow* and answers the following question:

> *Given a test suite, is it possible to reduce the time of the test suite execution process without any loss of information or a change on the underlying infrastructure?*

The answer to the above question is **yes** and we achieve it in two ways:

- By applying a data transformation to the *test inputs* and *test oracle* data of test suites in order to reduce the test suite compilation time without altering its semantics.
- By applying test case scheduling algorithms that reduce the overall test case execution time.

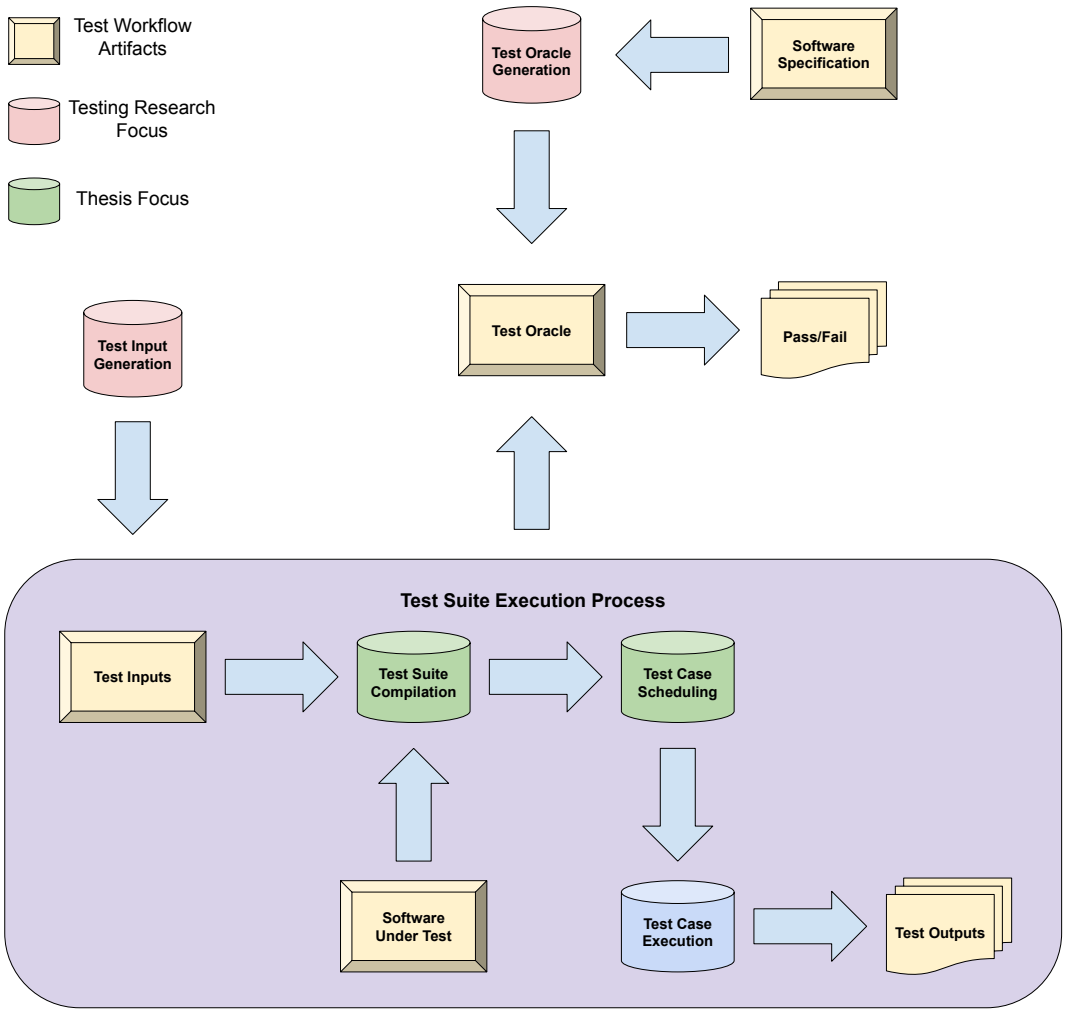

Figure 1.2: Granular Test Suite Execution Process

The main contributions of this thesis are:

- **C1: Data Transformation for Reducing Test Suite Compilation Time -** We propose a data transformation which is able to reduce the compilation time of test suites. We evaluate our approach on two popular compilers (GCC [55] and Clang [114]) compiling test suites from industry standard benchmarks as well as an industrial application developed by Codeplay Software [166]. The compilation speedup achieved ranges from 1.3x to 69x.

- **C2: Test Case Scheduling Algorithms for Improving Instruction Cache Locality -** We propose a series of test case scheduling algorithms for the single-CPU and multi-CPU architectures that enhance instruction cache locality **across** test case executions. We assess the effectiveness of these algorithms on vari-

ous industry standard benchmarks as well as on a module of the LLVM [115] tool-chain. The highest speedup achieved is 29.48%.

- **C3: Device-Based Test Case Scheduling for Heterogeneous Test Suites -** We propose a minimal-overhead test case scheduling algorithm for heterogeneous test suites which reduces the overall test suite execution time by achieving load balancing between the devices of the heterogeneous system the test suite executes on. This work was conducted in collaboration with Codeplay Software [166] for reducing the execution time of a large-scale industrial test suite targeting ComputeCPP™, Codeplay's in-house implementation of the SYCL standard [57]. The speedup achieved is 25.42%.

Figure 1.3 illustrates the thesis main contributions mapped to the test suite execution process. Contribution **C1** is applied just before the *test suite compilation* and results in time reduction for that phase. Contributions **C2** and **C3** however are applied during *test case scheduling* but result in time reduction for the next phase, the *test case execution*.

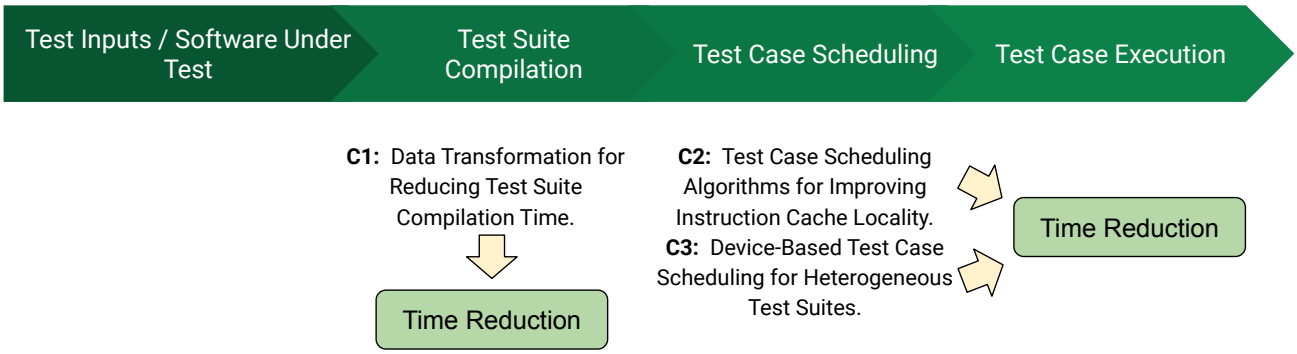

Figure 1.3: Thesis Main Contributions

1.5 Publications

- *Test case permutation to improve execution time* [188].
- *Improving test execution time with improved cache locality* [186].
- *Reordering tests for faster test suite execution* [189].
- *Assessing the Effect of Device-Based Test Scheduling on Heterogeneous Test Suite Execution* [187].

- *Speeding up test execution with increased cache locality* [190].
- *Assessing the effect of data transformations on test suite compilation* [191].

1.6 Organization

The rest of the thesis is organized as follows: Chapter 2 provides background on test suites and their optimizations as well as on the fundamental concepts on which the contributions of this thesis are based. In Chapter 3 we explore the related work for every main contribution of this thesis. Reducing test suite compilation time via data transformations is illustrated in Chapter 4. Our work on reducing test suite execution time with improved instruction cache locality is presented in Chapter 5. Chapter 6 addresses the problem of effective test case scheduling for heterogeneous software. Finally, this thesis concludes in Chapter 7.

Chapter 2

Background

In this chapter we provide the necessary background on the heterogeneous aspects and concepts of computer science that we utilise in this thesis for reducing test time. Sections 2.1 and 2.2 illustrate the foundations of test suites and their optimisations and relate to all three contributions of this thesis. In section 2.3 we present the concepts of compiler optimisation as well as data transformation that form the basis of our contribution **C1** which reduces test suite compilation time. We continue in section 2.4 by describing the notions of cache miss and cache locality which are essential for our test scheduling algorithms in contribution **C2**. Finally, in section 2.5 we give an overview of what heterogeneous computing is and how its different from sequential and multi-threaded computing when it comes to testing. The last section is related to our contribution **C3** where a test scheduling algorithm for heterogeneous applications is proposed.

2.1 The Anatomy of a Test Suite

As illustrated in the previous chapter, the software test *workflow* consists of four main *artifacts*: the *software under test*, its *specification*, the *test inputs* and the *test oracle*. But in which way are these *artifacts* reflected on actual code? How does an actual test suite looks like? In this section, we provide concrete examples for every test *workflow artifact* in the form of executable code in the C++ programming language [192] and the GoogleTest testing framework [175].

2.1.1 Software Under Test

The *software under test* can vary from a simple function to a complex application. In this chapter, our software under test will be a sorting algorithm: Listing 2.1 contains a sample implementation of the Bubble Sort algorithm [8]. The Bubble Sort algorithm is a well-studied sorting algorithm for single dimensional arrays. It works by repeatedly swapping the adjacent elements if they are in wrong order. In every iteration, the algorithm surfaces the maximum (or minimum - depending on the desired sorting order) element and then repeats the process for the remaining of the array. The *software under test* consists of two functions, the *bubbleSort* function which contains the sorting logic and the *swap* function that performs the swapping of two elements. In addition, our implementation contains a small optimization: if the algorithm does not swap any elements during an iteration, it automatically exits as this is an indication that the rest of the array is already sorted.

```
#include <vector>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of the Bubble Sort Algorithm.
std::vector<int> bubbleSort(int arr[], int n) {
   int i, j;
   bool swapped;

   if (n < 0){
      return std::vector<int>();
   }

   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
```

```
26              swap(&arr[j], &arr[j+1]);
27              swapped = true;
28          }
29      }
30
31      // IF no two elements were swapped by inner loop, then break
32      if (swapped == false)
33          break;
34    }
35
36    return std::vector<int>(arr, arr + n);
37 }
```

Listing 2.1: Bubble Sort Algorithm Implementation in C++

2.1.2 Software Specification

The *software specification* is a description of the software's intended behaviour and the basis of the *test oracle*. Despite the existence of international standards [174] which provide well-defined templates for expressing software requirements, the structure of this artifact can vary a lot across projects. There is a plethora of techniques and processes around *software specification* which cover all aspects of its life-cycle: from initial requirement gathering to validation and formatting [207]. The specification for our *bubbleSort* implementation is the following:

1. Should accept as inputs an array of integer values and the number of elements to be sorted.
2. Should return a vector of the sorted elements in ascending order.
3. If the given number of elements is a negative number, an empty vector should be returned.

2.1.3 Software Test Suite

Listing 2.2 contains a test suite for our *bubbleSort* implementation which is developed on top of the GoogleTest framework. It contains 8 test cases each of which is specified by the ***TEST*** keyword. Inside each test case, the *bubbleSort* function is executed, with a different *test input* every time, and the result its being passed into an assertion

function (***ASSERT_THAT***) which compares it against the expected value. Finally, on line 56, ***RUN_ALL_TESTS*** is responsible for scheduling the test cases for execution, executing them and returning the value ***0*** if every test case is successful (otherwise the value ***1*** is returned).

```
1  #include "bubbleSort.h"
2  #include <gtest/gtest.h>
3  #include <gmock/gmock.h>
4  /*
5    TEST(arg1, arg2) represents a single TEST CASE.
6      arg1: The test suite it belongs in.
7      arg2: The name of the TEST CASE.
8  */
9  TEST(bubbleSortTestSuite, SortsCorrectlyPositiveValues) {
10   int inputArray[] = {90, 64, 34, 25, 22, 12, 11}; // A TEST INPUT.
11   /*
12     ASSERT_THAT(arg1, arg2) represents an assertion by the TEST
         ORACLE.
13     arg1: An actual value obtained from the EXECUTION of PROGRAM
         UNDER TEST with these TEST INPUTS.
14     arg2: The expected value as defined by the TEST ORACLE.
15   */
16   ASSERT_THAT(bubbleSort(inputArray, 7 /*A second TEST INPUT.*/),
17               ::testing::ElementsAre(11, 12, 22, 25, 34, 64, 90));
18 }
19 TEST(bubbleSortTestSuite, SortsCorrectlyNegativeValues) {
20   int inputArray[] = {-34, -22, -100, -201, -3, -1, -2909, -512};
21   ASSERT_THAT(bubbleSort(inputArray, 8),
22               ::testing::ElementsAre(-2909, -512, -201, -100, -34,
                   -22, -3, -1));
23 }
24 TEST(bubbleSortTestSuite, SortsCorrectlyMixedValues) {
25   int inputArray[] = {-200, 432, -43, 2, 0, -200, 0, 2};
26   ASSERT_THAT(bubbleSort(inputArray, 8),
27               ::testing::ElementsAre(-200, -200, -43, 0, 0, 2, 2,
                   432));
28 }
29 TEST(bubbleSortTestSuite, SortsCorrectlyPartialArray) {
30   int inputArray[] = {64, 34, 25, 12, 0, 11, 90, -20, 1000, -21};
31   ASSERT_THAT(bubbleSort(inputArray, 5),
32               ::testing::ElementsAre(0, 12, 25, 34, 64));
```

```
33 }
34 TEST(bubbleSortTestSuite, EmptyArray) {
35   int inputArray[] = {};
36   ASSERT_THAT(bubbleSort(inputArray, 0),
37               ::testing::ElementsAre());
38 }
39 TEST(bubbleSortTestSuite, SingleElementArray) {
40   int inputArray[] = {43};
41   ASSERT_THAT(bubbleSort(inputArray, 1),
42               ::testing::ElementsAre(43));
43 }
44 TEST(bubbleSortTestSuite, AlreadySortedArray) {
45   int inputArray[] = {1, 34, 56, 67, 89, 123, 456, 981, 2024};
46   ASSERT_THAT(bubbleSort(inputArray, 9),
47               ::testing::ElementsAre(1, 34, 56, 67, 89, 123, 456,
                    981, 2024));
48 }
49 TEST(bubbleSortTestSuite, NegativeElementNumber) {
50   int inputArray[] = {64, 34, 25, 12, 0, 11, 90, -20, 1000, -21};
51   ASSERT_THAT(bubbleSort(inputArray, -5),
52               ::testing::ElementsAre());
53 }
54 int main(int argc, char **argv) {
55   ::testing::InitGoogleTest(&argc, argv);
56   return RUN_ALL_TESTS(); // TEST CASE SCHEDULING and EXECUTION.
57 }
```

Listing 2.2: Bubble Sort Test Suite in C++ and Google Test

2.1.3.1 Test Inputs

Every test case in the test suite of listing Listing 2.2 contains a *test input* which is passed to the *bubbleSort* function. The *software specification* of our implementation requires that the *bubbleSort* function accepts an array of integer as well as the number of elements of that array which should be sorted. Therefore, the *test input* of each test case consists of an array of integer values as well as an integer scalar. In every test case the input array is being stored to a separate variable every time (*inputArray*) while the integer scalar is being passed directly to the *bubbleSort* function (second argument). All the *test inputs* of the *bubbleSort* test suite have been manually implemented by the developer. In the last decades, there has been a lot of attention by the scientific

community on automated *test input* generation [6] based on the *software specification* as well as the application program interface (API).

A careful reader would notice that some of the *test inputs* differ quite significantly across test cases. For example the test case of line 9 contains an *inputArray* with only positive values while the *inputArray* in line 19 contains only negative values. This is because our test suite attempts to verify that *bubbleSort* would behave in accordance to its *specification* for a variety of scenarios. The test cases in lines 9, 18 and 24 verify that *bubbleSort* can sort arrays with elements of any sign. The test case in line 29 tests that *bubbleSort* is able to sort only a part of the input array as defined in the first *specification*. The test case in line 49 tests the third *specification* which dictates that *bubbleSort* should return an empty vector if the number of elements to be sorted is a negative number. Finally, the test cases of lines 34, 39 and 44 test some edge cases (empty array, single-element array and already sorted array).

2.1.3.2 Test Oracle

In every test case of our test suite, ***ASSERT_THAT*** accepts two arguments: The first is the **actual** return value of *bubbleSort* for the specific *test input*. The second is the **expected** return value of *bubbleSort* given this *test input*. This value is defined by the *test oracle*. In our case, the *test oracle* is the developer who manually implemented these tests by consulting the *software specification*. In other words, the *test oracle* is hard-coded in the test code itself. Automatically generating the *test oracle* from the *software specification* is a field of active research in recent years [176]. Finally, in our test suite, the *test oracle* observes **only** the return value of the *bubbleSort* function. This kind of testing is defined as *black-box* testing [12] because we are only interested in the output of the *software under test* for a given *test input*. In *white-box* testing [154], the *test oracle* can also observe the internals of the software execution.

2.1.3.3 Test Case Scheduling

Test case scheduling is the phase where the testing framework produces the test case *execution order*. The *execution order* type can vary depending on the underlying machine architecture. For example, in single-CPU systems the *execution order* can only be *sequential* while in multi-CPU it could also be expressed in terms of test case *clusters* that should execute in parallel. In our test suite, the test case scheduling happens in line 56: the first step of ***RUN_ALL_TESTS*** is to produce the *execution order* of

the test cases before it proceeds to the actual test case execution and, subsequently, to the result report. By default, ***RUN_ALL_TESTS*** produces only *sequential execution orders* - however the GoogleTest framework can be customized to allow parallel test case execution.

2.2 Test Suite Optimizations

Over the last decades, the research community has focused on discovering ways for reducing the number of test cases. Code coverage metrics are being used by industry and academia to describe the degree to which a program is tested by a test suite. These criteria are frequently encountered in regression [169] and black-box [12] testing where the number of test cases can become intractable for non-trivial software. Numerous optimisation techniques have been proposed which, based on coverage criteria, attempt to reduce the number or the size of the test cases. We start this section with an introduction to the foundation of coverage analysis, the *control flow graph*. We continue by providing the definitions of the *statement* as well as *branch* coverage metrics. Finally, we explore the main test suite optimisation techniques for reducing the effort every time a test suite is executed.

2.2.1 Control Flow Graph

In control flow analysis [4], the subject program is expressed as a directed graph (the Control Flow Graph - CFG) with the nodes representing the programs *basic blocks* and the edges representing the control flow paths. A *basic block* is defined as a linear instruction sequence with a single entry point (the first executed instruction) and a single exit point (the last executed instruction). From the moment the first instruction of a *basic block* is executed, it is guaranteed that all the other instructions of that *basic block* will be executed as well (i.e. the control flow does not diverge while inside a basic block). CFG's are being used extensively in compiler engineering as well as in coverage analysis: Expressing a program as a graph, enables the application of *graph theory* concepts, such as *reachability* and *domination* relationships, directly onto the program instructions. In fact, many compiler optimizations are based on *graph theory* concepts [3]. Furthermore, as illustrated in the next sections, most coverage metrics are defined on top of the CFG.

Figure 2.3 contains the CFG for our *bubbleSort* function. Our function entry point

is ***BBL_1*** and it is the only *basic block* that does not have a predecessor. Similarly, the *bubbleSort* function contains two exit points which are represented by two terminating basic blocks with no successors - ***BBL_2*** and ***BBL_4***. The diamond shapes represent decision statements (or decision points). Every decision point consists of the beginning of two branches corresponding to the two possible decision outcomes: true and false. Finally, its worth noting that although the individual program instructions can be easily mapped to the program's original source code, other language constructs like *loops* and *switch* statements are expressed in an indirect way via branches and decision statements.

2.2.2 Statement Coverage

Statement coverage of software P against a test suite T is defined as the number of software instructions of P which have been executed at least once during the execution of the test suite T divided by the total number of instructions of program P:

$$Statement\_Coverage(P,T) = \frac{\#instructions\ of\ P\ executed}{\#total\ instructions\ of\ P} \tag{2.1}$$

Statement coverage can also be equivalently expressed in terms of visited *basic blocks*. In fact, most program analysis and optimization algorithms usually work with *basic blocks* rather than instructions for scalability reasons.

$$Statement\_Coverage(P,T) = \frac{\#basic\ blocks\ of\ P\ visited}{\#total\ basic\ blocks\ of\ P} \tag{2.2}$$

Starting from an empty test suite, if we add the test case in line 49 of listing 2.2 (*NegativeElementNumber*) we achieve 18.18% statement coverage: this is because we have visited only the ***BBL_1*** and ***BBL_2*** *basic blocks* (2 out of the 11). If we then continue by adding the test case in line 34 (*EmptyArray*) to our new test suite we immediately double our statement coverage (4 out of the 11 or 36.36%) because ***BBL_3*** and ***BBL_4*** will be visited as well. Having the test case in line 24 (*SortsCorrectlyMixedValues*) as our third test case helps us achieve full statement coverage as every *basic block* will be visited at least once during the execution of our new test suite.

2.2.3 Branch Coverage

Branch coverage aims to ensure that every possible branch from every decision point in the *software under test* has been executed at least once during the execution of a test suite. Branch coverage is defined as the number of decision outcomes which have

occurred at least once during the execution of a test suite divided by the total number of the programs decision outcomes:

$$Branch\_Coverage(P,T) = \frac{\#decision\ outcomes\ of\ P\ occurred}{\#total\ decision\ outcomes\ of\ P} \tag{2.3}$$

Starting again from an empty test suite, if we add the test cases of lines 49 and 34 of listing 2.2 (*NegativeElementNumber* and *EmptyArray*) we achieve 30% of branch coverage because the ***B1***, ***B2*** and ***B4*** branches have been taken at least once (3 out of 10). In other words, with these two test cases, both the decisions outcomes of the first decision point ($n < 0$) and the negative outcome of the second decision point ($i < n-1$) have occurred. By adding the test case in line 9 (*SortsCorrectlyPositiveValues*) our branch coverage scales to 80%: the positive outcome of the second decision point occurs (***B3***), both outcomes of the $j < n-i-1$ decision point occur (***B5*** and ***B6***), the positive outcome of the $arr[j] > arr[j+1]$ decision point occur as well as the negative outcome of the $swapped == false$ decision point. For achieving full branch coverage we need a test case in which the $arr[j] > arr[j+1]$ decision point would evaluate to false (***B10***). We also need a test case where our algorithm optimisation as described in section 2.1.1 (line 34 of listing 2.1) is triggered (***B7***). The *AlreadySortedArray* test case (line 44 in listing 2.2) satisfies both these criteria because it guarantees that no element swapping will take place: branch ***B10*** will be taken in every iteration of the inner loop and branch ***B7*** will be taken after the first iteration of the outer loop.

2.2.4 Test Suite Minimization

Test suite minimisation [215] refers to the systematic removal of test cases while ensuring that the test suite satisfies a set of requirements. If the requirement is full branch coverage, test suite minimization algorithms will keep in the test suite the minimum number of test cases that achieve 100% branch coverage and **remove** the rest. In the previous section, we identified that 4 test cases from listing 2.2 achieve full branch coverage (*NegativeElementNumber*, *EmptyArray*, *SortsCorrectlyPositiveValues* and *AlreadySortedArray*). Most test suite minimisation algorithms would be in a position to identify that the *EmptyArray* test case is actually redundant and can be removed without impacting our test suite's branch coverage. Therefore a potential form of our minimised test suite would be the following 3 test cases: *NegativeElementNumber*, *SortsCorrectlyPositiveValues* and *AlreadySortedArray*. All other test cases would have been removed and never executed.

2.2.5 Test Case Selection

Test case selection [152] is less aggressive than test suite minimization of section 2.2.4. Instead of removing test cases, it selects a subset of them for execution according to a criterion of interest. The set of selected test cases might be different every time the test suite is executed. For example, in a system comprising of multiple sources files, a common test case selection criterion is to select for execution *only* the test cases which exercise code from sources files that have been *modified* since the last time the test suite was executed. If a test case itself has been modified then its selected for execution regardless. By this way, we avoid executing test cases for which we are certain that their outcome will not change when compared to their last execution because their code as well as the code which they verify has not changed.

2.2.6 Test Case Prioritization

Once the test cases that need to be executed have been selected (see section 2.2.5), test case prioritization [42] will rank them based on how much they contribute towards achieving a certain criterion, such as branch coverage. Subsequently, these test cases will be *ordered* for execution according to their rank: the test case with the highest rank will be executed first, the test case with the second highest rank will follow etc. Continuing our example from section 2.2.4, the 3 test cases achieving full branch coverage would get a higher rank than the other test cases and would be executed first every time the *full* test suite is executed. The main idea behind test case prioritization is that the subset of test cases satisfying a set of requirements (in our example branch coverage) is executed first in order for these requirements to be satisfied as soon as possible during the execution of a test suite.

2.2.7 Test Case Reduction

Test case reduction [95] examines each test case in isolation and attempts to remove redundant behaviour (i.e. preserve the semantics of the test without performing any additional tasks). Listing 2.3 contains an example of test case reduction: From a branch coverage point of view, *AlreadySortedArray* and *AlreadySortedArrayReducted* visit the exact same branches, therefore using the reducted version is preferred as it executes faster than the original version (in this case because of fewer algorithm iterations as the input array is smaller in size).

```
// Original Test Case
TEST(bubbleSortTestSuite, AlreadySortedArray) {
  int inputArray[] = {1, 34, 56, 67, 89, 123, 456, 981, 2024};
  ASSERT_THAT(bubbleSort(inputArray, 9),
              ::testing::ElementsAre(1, 34, 56, 67, 89, 123, 456,
                  981, 2024));
}
// Reducted Test Case
TEST(bubbleSortTestSuite, AlreadySortedArrayReducted) {
  int inputArray[] = {1, 34};
  ASSERT_THAT(bubbleSort(inputArray, 2),
              ::testing::ElementsAre(1, 34));
}
```

Listing 2.3: Test Case Reduction

2.3 Compiler Optimizations

In contribution **C1** we reduce the compilation time of test suites by applying data transformations. More specifically, our proposed data transformation speeds-up various optimizations being performed by the compiler resulting in an overall time reduction of test suite compilations. In this section, we provide the necessary background on compiler optimizations as well as data transformations.

Compiler optimizations [3] consist of transformation algorithms that produce a semantically equivalent version of a given program, optimized in certain ways – typically to reduce execution time and/or memory operations. In the present thesis, we use C-language family compilers which implement the following optimization options:

- **-O0** Optimizations are disabled.
- **-O1** Moderate optimization. Compilation takes more time for large functions.
- **-O2** High optimisation. All supported optimizations that do **not** involve a space-speed trade off are performed. Compilation time significantly increases.
- **-O3** Full optimisation. All possible optimisations are performed. Compilation time rapidly increased.

2.3.1 Data Transformations

Data transformations are amongst the most common *optimizing transformations* that compilers utilize in order to optimise. Data transformations are defined by Boyle et al. [156] as "those transformations concerned with the layout, storage and access of array data, rather than reordering the program control flow". Figure 2.1 illustrates an array data transformation: Before the transformation, the data is stored in a two-dimensional array and every value needs two indexes in order to be retrieved (for example, the value 7 can be retrieved by indexing the table with the indexes 0 and 2). After the transformation, the data layout has been changed into a one-dimensional array and every value needs just one index in order to be retrieved (for our previous example, number 7 now can be retrieved with the index 6). Note that there is no data loss during the transformation! Only the layout has changed.

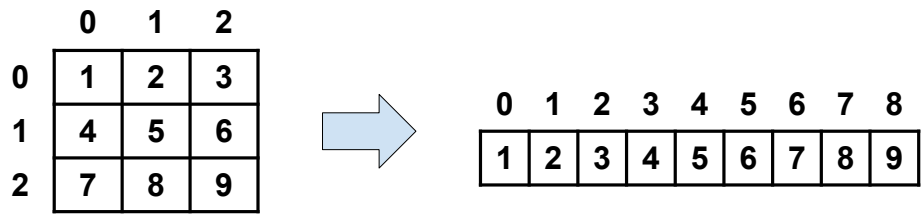

Figure 2.1: Two-Dimensional to One-Dimensional Data Transformation of an Array

2.4 Cache Memory

In contribution **C2** we achieve a reduction in test suite execution time by improving the instruction cache locality via test case scheduling. Increased cache locality leads to a decrease of cache misses which, in turn, leads to a reduction of the overall test suite execution time. In this section we illustrate the concepts of cache memory, cache misses as well as cache locality.

Present day modern applications require a vast amount of memory in order to meet their data requirements. Additionally, processor speeds have become much faster than memory speeds. As a result, execution times of many applications are memory speed, rather than processor speed bounded [58]. To help bridge the speed gap, memory systems are organized as a hierarchy with multiple layers of fast cache memory. CPU caches comprise of an *instruction cache* to speed up executable instruction fetch and a *data cache* to speed up data fetch and store. Caches play a key role in minimizing

the access latency and main memory bandwidth demand. Caches operate by retaining the most recently used data. If the processor reuses the data quickly, cache hits occur. Conversely, if it reuses the data after a long time, intervening data can evict the data from the cache, resulting in a cache miss. Instruction cache misses and hits are illustrated in Figure 2.2. Cache misses cause the CPU to stall and in many applications result in significant penalty in execution time [199, 203].

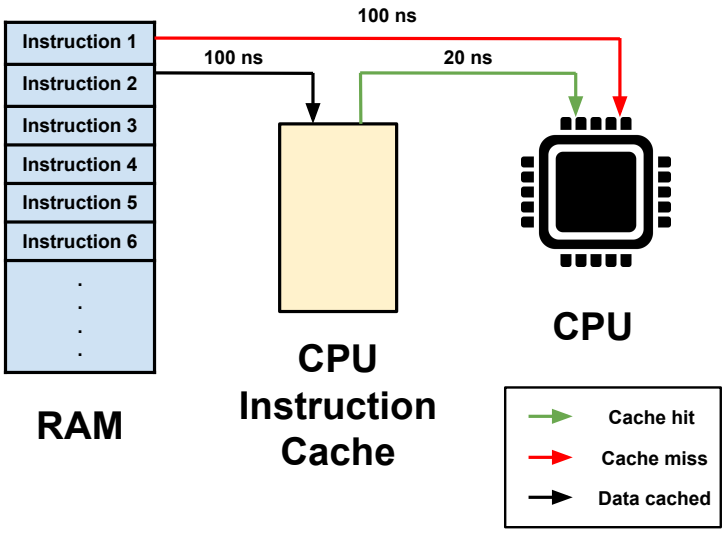

Figure 2.2: CPU Instruction Cache

2.4.1 Cache Locality

Cache misses have been shown to be inversely proportional to the locality of memory references during program execution [53]. **Temporal locality** is achieved by minimizing the time between references of the same memory address, i.e. the reuse of the same data within a small time frame. **Spatial locality**, on the other hand, is achieved when memory accesses which are close in time are also close in physical storage location. In terms of cache memory design, fetching large blocks of data (cache lines) when a cache miss occurs has the potential to increase spatial locality. However, this approach may have a negative effect on temporal locality since there is no guarantee that the additional data of the fetched cache line will be useful. Predictors and pre-fetch buffers have been proposed by the research community in a attempt to improve spatial locality without compromising temporal locality [120].

2.5 Heterogeneous Computing

In contribution **C3** we propose a test case scheduling algorithm for heterogeneous applications which reduces the overall execution time of test suites by achieving load balancing between the devices of the heterogeneous system. In this section we provide an introduction to heterogeneous computing and the challenges around testing heterogeneous applications.

Current semiconductor trends show a major shift in computer system architectures towards heterogeneous systems. Such systems combine a CPU with other processors such as GPUs, DSPs and FPGAs to work together, performing different tasks in parallel. This shift has brought a dramatic change in programming paradigms and languages. That has resulted in not only changes in the way that applications need to be structured, but also in the methods and requirements for testing them.

The main motivation for heterogeneous computing is the observation that certain processor types are better suited for certain types of computational tasks. For example, CPUs are great for performing very complex calculations but they cannot offer task parallelization on a massive scale. On the other hand, GPUs can execute a massive number of computational tasks in parallel, the complexity of these tasks however is limited. Over the last decades, heterogeneous computing has been of great importance for multimedia and gaming applications as well as 3D modelling and simulation software. Additionally, in recent years, heterogeneous systems are used extensively for accelerating deep learning applications [1] and provide the cornerstone of self-driving cars [94].

2.5.1 The Anatomy of a Heterogeneous Application

A heterogeneous application can be divided in two main parts - the code executing on *host* and the code executing on *device*. In most heterogeneous programming models, the *host* refers to the CPU and the memory of the machine the application is executing on while *device* refers to the heterogeneous devices that are available to the system (GPUs, other CPUs, FPGAs, DSPs etc). The *host* code can manage the *host* memory as well as the memory of all the available *devices*. For executing computational tasks, the *host* submits to the *devices* functions to be executed (aka *kernels*) along with data and then proceeds with other computations while the *devices* are executing the *kernels* (i.e. *asynchronous* programming).

Listing 2.4 includes a sample heterogeneous application developed by Codeplay

Software ltd. [166] in C++ and the SYCL [57] platform. The purpose of this application is to add, in parallel, the elements of 2 vectors. *simple_vadd* accepts 3 vectors as arguments: 2 of them represent the vectors to be added and the third vector is used for storing the result. *simple_vadd* starts by defining a *queue* (the mechanism for submitting *kernels* to be executed by the *devices*) and also defines a series of *buffers* for making available the vector data to the *device*. Lines 39 to 51 contain the *device* code: The *device* uses the *buffers* to access the vector data and on line 49 the parallel addition of the vector elements occurs.

```
1  /*
2   *
3   *  Copyright (C) 2016 Codeplay Software Limited
4   *  Licensed under the Apache License, Version 2.0 (the "License");
5   *  you may not use this file except in compliance with the License.
6   *  You may obtain a copy of the License at
7   *
8   *      http://www.apache.org/licenses/LICENSE-2.0
9   *
10  *  Codeplay's ComputeCpp SDK
11  *
12  *  simple-vector-add.cpp
13  *
14  *  Description:
15  *    Example of a vector addition in SYCL.
16  *
17  */
18 #include <CL/sycl.hpp>
19 #include <array>
20 #include <iostream>
21
22 constexpr cl::sycl::access::mode sycl_read = cl::sycl::access::mode
        ::read;
23 constexpr cl::sycl::access::mode sycl_write = cl::sycl::access::mode
        ::write;
24
25 template <typename T>
26 class SimpleVadd;
27
28 template <typename T, size_t N>
29 void simple_vadd(const std::array<T, N>& VA,
30                  const std::array<T, N>& VB,
```

```
                    std::array<T, N>& VC) {
  cl::sycl::queue deviceQueue;
  cl::sycl::range<1> numOfItems{N};
  cl::sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);
  cl::sycl::buffer<T, 1> bufferB(VB.data(), numOfItems);
  cl::sycl::buffer<T, 1> bufferC(VC.data(), numOfItems);

  deviceQueue.submit([&](cl::sycl::handler& cgh) { // DEVICE CODE.
    // Accessing the input arrays from the DEVICE.
    auto accessorA = bufferA.template get_access<sycl_read>(cgh);
    auto accessorB = bufferB.template get_access<sycl_read>(cgh);
    auto accessorC = bufferC.template get_access<sycl_write>(cgh);

    auto kern = [=](cl::sycl::id<1> wiID) {
      accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
    };
    // Parallel addition of vector elements.
    cgh.parallel_for<class SimpleVadd<T>>(numOfItems, kern);
  });
}

int main() {
  const size_t array_size = 4;
  std::array<cl::sycl::cl_int, array_size> A = {{1, 2, 3, 4}}, B =
      {{1, 2, 3, 4}}, C; // Sample Vectors to be added.
  simple_vadd(A, B, C); // Parallel vector addition.
  return 0;
}
```

Listing 2.4: Heterogeneous Implementation of Parallel Vector Addition

2.5.2 Testing Heterogeneous Software

One of the major complications when testing a heterogeneous system is that much consideration should be put on the hardware on which the tests are executed. In a typical business case scenario, it is **crucial** for a heterogeneous system test suite to be able to verify that the system works as expected when executed against a variety of *devices*. Given this requirement, the problem of executing N tests on a sequential or multi-core system is **expanded** to the execution of $N * M$ tests for a heterogeneous system, with M being the number of *devices* for the heterogeneous system to be verified against.

The rational behind this expansion is the fact that the *device* code of a heterogeneous application needs to be executed on **every** target *device* before it is considered fully tested and ready for release.

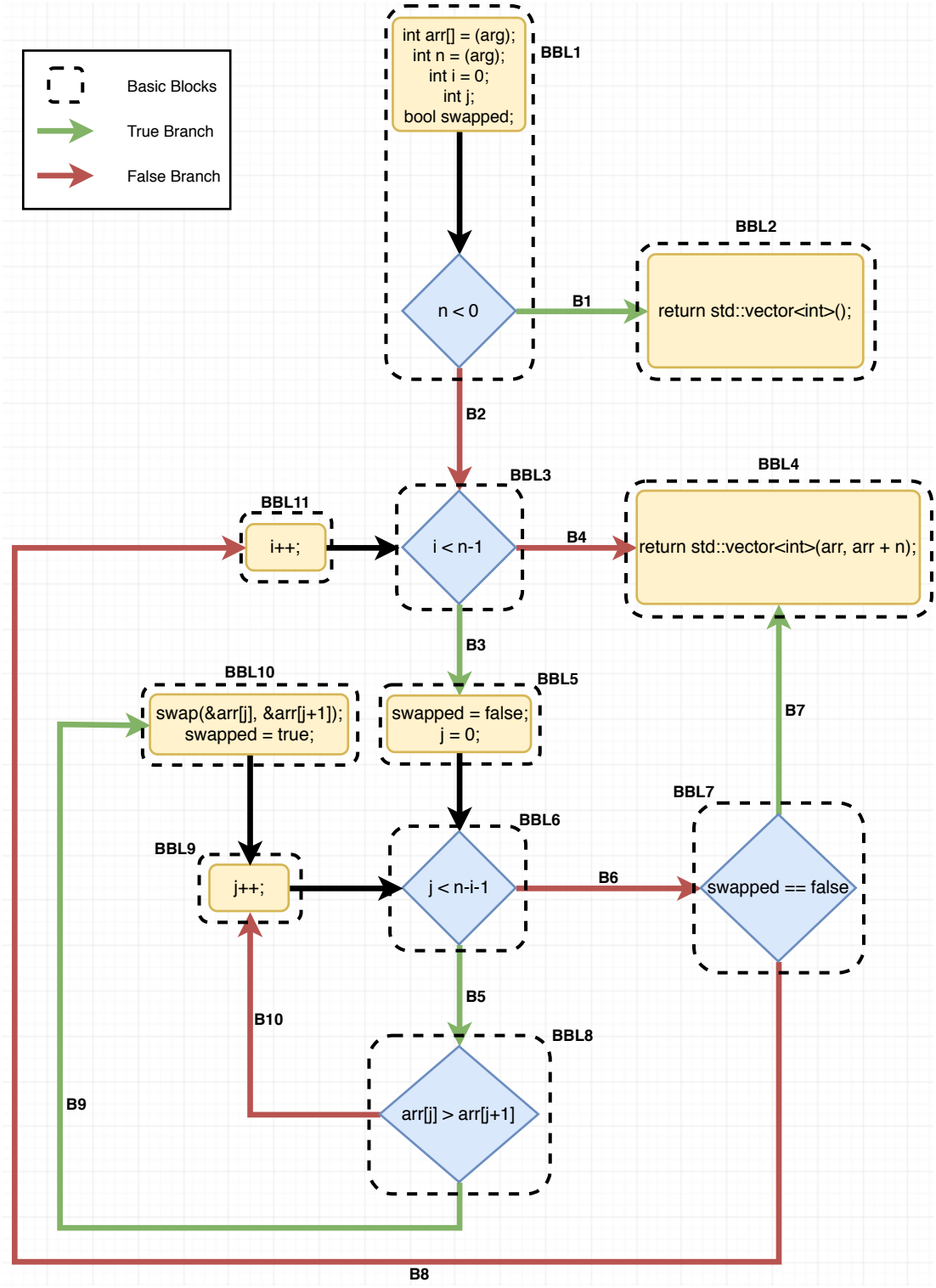

Figure 2.3: Bubble Sort Implementation Control Flow Graph

Chapter 3

Related Work

This Chapter includes the related work for every contribution of this thesis. We begin with the state of the art on compilation time reduction techniques (section 3.1) and then we continue by presenting the foundations as well as applications of data transformations and parameterised unit tests in section 3.2. Section 3.3 includes the related work on Testability Transformation as well as its key differences when compared to the data transformation we propose in contribution **C1**. In section 3.4 we present the related work on improving cache locality and how it can be used in conjunction with our work in contribution **C2**. Test Case Selection techniques are presented in section 3.5. We proceed by illustrating the related work on Test Case Prioritization and the subtle differences of traditional *TCP* techniques when compared to our **C2** and **C3** contributions. Work on executing test cases in parallel is considered in section 3.7. Finally, section 3.8 includes the existing work on system-on-a-chip test scheduling and its limitations when it comes to heterogeneous applications - something which we address in contribution **C3**.

3.1 Reducing Compilation Time

For trivial programs, compilation time is insignificant, but quickly increases as programs become more complex. Performing multiple compiler optimisations adds significant overhead to compilation time. Reducing compilation time is an important problem that has been addressed in several ways,

- The C++ programming language has introduced the *zero overhead principle* which dictates that no overhead, both during compilation and execution, should occur for

features of the language that are not being used [192]. Furthermore, the GNU compiler collection [184] has introduced in its C/C++ compilers the *-O1* optimization level which includes only lightweight optimizations that do not result in long compilation times.

- Krintz et al. [107] propose an annotation framework for Java programs which collects off-line analysis information and embeds it, in the form of annotations, into Java programs in order to guide the optimization process of dynamic compilers, reducing compilation overhead. In [108] Krintz et al. present the concept of *lazy compilation* in which a method is compiled just before its first invocation and then augment this concept by exploiting profiling information to ensure that performance critical methods are invoked using optimized code.

- When compiling for FPGAs, Lavin et al. [116] propose the use of pre-compiled circuit blocks, known as *hard macros*, as a way to speed up the compilation process. Chan et al. [23] present a compilation time reduction scheme which is based on SAT engine partitioning in order to reduce the compilation time of the FPGA-based SAT solver presented in [225].

- Machine learning techniques have also been proposed for reducing compilation time. Cavazos and O'Boyle [19] propose the use of logistic regression for building a probabilistic model in order to select the best optimizations per method in Java programs while Leather et al. [117] introduce a mechanism to automatically identify the important features of programs that can be used by machine learning heuristics.

- Iterative compilation, which is proposed by Kisuki et al. in [104] and evaluated by Fursin et al. in [49], is a method in which successive source-to-source transformations are applied to a program. Their impact is determined by compiling and executing the code. This results in multiple versions of the program with the best version being picked based on criteria of compile and/or execution time.

Our approach in contribution **C1** reduces compilation time of test code by applying source-to-source transformation before compilation takes place. It is similar to *iterative compilation* methodologies in that it includes source-to-source transformations as a pre-compilation step. The main drawback of iterative compilation is its feasibility, as even with a small set of possible code transformations, the resulting optimization space is very large. This is addressed in the works of Fursin et. al. [49],Bodin et. al. [15] and Triantafyllis et. al. [200] which propose ways to reduce the search space by

utilizing heuristics. In contrast, our approach focuses on exploiting a common pattern for calling test functions and is able to use a *single* source-to-source transformation in order to reduce compilation time in test code.

3.2 Data Transformations and Parameterized Unit Tests

In contribution **C1**, we apply a *data transformation* to achieve reduction in test code compilation time. In [156], Boyle et al. define and validate an algebraic framework for data transformations in which an array transformation consists of a change in the way it is stored and accessed. Data transformations have been subsequently explored for various purposes. Kandemir et. al. in [99] and [98] as well as Rivera et. al. in [167] utilize data transformations in order to improve cache memory locality. In [89], data transformations are used for reducing the number of false sharing misses in a shared memory multiprocessing system and in [87], they are used for enabling loop vectorization on data parallel architectures. To the best of our knowledge, there has been no prior work exploring data transformations to reduce compilation time of test code. In our approach, we apply data transformations that target test code compilation.

Our *data transformation* is applied to *parameterised unit tests (PUTs)*, introduced by Tillman and Schulte in [198] and also used in commercial test frameworks like Google Test [175]. PUT extends conventional unit tests by allowing the user to parameterize them and generate multiple traditional unit tests from a single PUT. In this way, PUTs are used for test generation. The approach employs symbolic execution for systematically producing a minimal set of parameters which results in the generation of a set of concrete tests that execute a finite number of paths in the system under test. Our proposed transformation is applicable on the concrete tests that have been generated from PUTs for reducing their compilation time.

3.3 Testability Transformation

Testability transformation is a source-to-source transformation of programs that enables test generation methods to produce more effective test data for the original (pre-transformed) program. After the test data has been generated, the transformed program is discarded. A testability transformation does not guarantee functional equivalence: it only preserves the test adequacy of test input data sets. For example, if a test data set

achieves 100% branch coverage in program P, it is guaranteed that the *same* test data set will achieve the same level of branch coverage for the transformed program P'.

- Harmal et. al. introduce the theoretical framework of testability transformation in [66] and propose a testability transformation algorithm that replaces boolean flags with comparison expressions for improving the effectiveness of evolutionary test data generation in [10]. An improvement of the flag removal algorithm along with a tool is presented in [14].

- Evolutionary test data generation is also considered in [136] where the authors empirically evaluate a transformation algorithm that removes nested decision statements.

- The relationship of refactoring methods to testability transformation is explored by Harmal et. al in [65].

- Hierons et. al. propose a testability transformation aimed at unstructured programs that removes multiple *exit* statements inside *for* loops while preserving branch coverage [77].

- In [97] Kalaji et. al. classify program functions depending on their relationship to global state variables (*affect* or *affected-by*) and propose a testability transformation which surfaces the conditions of these global variables that need to hold in order for specific parts of the program to be executed.

- Korel et. al. propose in [105] a testability transformation which is based on data dependence analysis and preserves *only* the statements that contribute to the calculation of the test case generation *fitness* function.

- Using testability transformations for generating pseudo-oracles is considered by McMinn in [135]. In his work, McMinn alters specific aspects of programs (one transformation per aspect) and then uses search based testing in order to identify test cases where the output of the transformed versions is different to the one of the original program. This methodology differs from traditional testability transformations in that it is important that the transformed programs are functionally equivalent to the original version.

- Li and Fraser propose in [123] another testability transformation that preserves the semantics of the original program. They target the boolean flag problem

([10], [14]) and apply their transformation to Java bytecode. The main idea of the transformation algorithm is to replace all boolean variables with integers that preserve *branch distance* information.

Our data transformation in Chapter 4 differs from testability transformation techniques in the fact that it operates on the test suite rather than the software under test. Furthermore, the transformed test suite is not discarded and it actually replaces the original. Finally, our data transformation preserves the semantics of the original test suite, something which is not the case with the majority of testability transformations.

3.4 Improving Cache Locality

Improving data locality is a problem which has received great attention by the compiler community in the last decades. Compiler researchers have proposed the use of reuse distance as a metric to approximate cache misses [13, 162]. Beyls et al. state reuse distance of a memory access as "the number of accesses to unique addresses made since the last reference to the requested data". In a fully associative cache with n lines, a reference with reuse distance $d < n$ will hit, and with $d \geq n$ will miss. The concept of cache re-use has primarily been used in the context of data locality.

In the early 1990s, compiler optimisations were proposed to improve the cost of executing loops [208, 17]. These optimisations improve locality of data references in loops through:

- **Loop permutations** - If possible, change the sequence of loop iterations so that the iteration which enhances data reuse is placed innermost [134].
- **Loop tiling** - Iterations are reordered so that outer loop iterations are executed without waiting for the iterations of inner loops to complete execution. By this way, the distance reuse of data associated with the outer loops is decreased.
- **Loop fusion** - Multiple loops are merged under one.
- **Loop distribution** - Independent statements inside a single loop are separated into multiple, single-statement loops.
- **Variable padding** - Inter-variable padding refers to adjusting variable base addresses while intra-variable padding refers to modifying the size of data arrays.

Both techniques are used heavily in compilers and have been identified to be effective in minimizing conflict misses in loops [134].

Procedure re-ordering and code layout optimisations are available in the literature for improving instruction spatial locality. Chang et al. [24] use dynamic profiling in conjunction with function inlining in an attempt to position instructions in such a way that spatial instruction locality is maximised. Chen et al. [26] propose a co-location technique for functions and basic blocks which are visited sequentially for achieving greater spatial locality.

Temporal locality of instructions has not been considered before, especially since existing optimisations are over a *single execution* of the program with little chance of repeated instruction sequences[1]. Temporal locality across multiple executions is proposed, for the first time, in contribution **C2**. Our approach presented in chapter 5 is not meant to compete with the existing work on compiler or code layout optimisation. Instead, it is best if they are used **together** since we aim to improve *temporal* locality of *instructions* across several executions, while existing work improves temporal/spatial locality of *data* and *spatial* locality of *instructions* within a single execution.

3.5 Selecting Test Cases

Test case selection is a test suite optimization that reduces the number of executed tests cases during a test suite execution by selecting a subset of them based on some criterion. It was first proposed by Fischer et. al. in [48] and it is closely related to *regression testing* since it determines *which* test cases *have* to be executed during the execution of a test suite. Test case selection strives to achieve a trade-off between the cost of executing all the test cases and the probability of not executing a test case that would uncover a bug in the software.

Effective test case selection can reduce the cost of testing without sacrificing the fault-finding capability of test suite executions and has been the center of attention by industry and academia for over four decades. Table 3.1 includes the (broad) technique types that test case selection studies have been based on.

Our proposed methodologies in Chapters 5 and 6 do not entail any test case selection techniques. However, both our contributions can be used in conjunction with test case selection: Once the subset of test cases has been chosen to execute (based

[1] Unless the instructions occur within a for loop, in which case existing loop transformations help improve locality.

on some criterion) then our test scheduling algorithms can speed-up the test case execution for CPU-based (contribution **C2**) as well as heterogeneous (contribution **C3**) architectures.

Technique Type	Studies
Test Coverage	[210], [204], [79], [69], [68], [47], [28], [2], [201], [34]
Control Flow Graph	[206], [168], [40], [121]
Program Slicing	[59], [29], [195], [212]
Machine Learning	[25], [219], [158]
Test Oracle	[220]
Genetic Algorithms	[71], [78], [113]
GUI	[139], [140], [141]
Multi-Objective Optimizations	[215], [144], [159]
Model-Based	[196], [211], [72], [73], [85], [131]
Binary Code Changes	[80]
Dynamic Non-Code Changes	[54], [151]
Specification-Based	[27]
Meta-Heuristics	[150]
Fuzzy Reasoning	[213], [214], [112]
Component-Based	[132], [157], [172]

Table 3.1: Test Case Selection Studies Grouped by Technique Type

3.6 Prioritizing Test Cases

Test case prioritization is a test suite optimization that creates test case execution orderings based on some objective function. Once the test cases to be executed have been selected (see section 3.5) test case prioritization will order them according to their importance defined by the objective function. The core idea of test case prioritization is that the most important test cases are executed first so that a specific criterion is met as soon as possible. This is particularly important for large and time consuming test suites where the full completion of a test suite might not be feasible or guaranteed. Similarly to test case selection, test case prioritization is closely related to *regression testing* and it was first proposed by W. Eric Wong et. al. in [209]. Table 3.2 includes

the main technique types that test case prioritization studies have been based on over the last decades.

Technique Type	Studies
Test Coverage	[170], [33], [34], [64], [63], [223], [142], [45], [16], [95], [119], [109], [93], [88]
Search-Based	[124], [125], [118], [91], [39]
Genetic Algorithms	[130], [127], [222], [18], [101], [96], [171], [31], [37], [178], [50], [155]
Requirement-Based	[180], [149], [129], [7], [75], [217], [181]
History-Based	[155], [179], [126], [133], [102], [103], [83]
Risk-Based	[75], [217], [76], [216], [182], [46]
Bayesian Networks	[202], [224], [145], [146]
Multi-Objective Optimizations	[173], [193]
Model-Based	[197], [106]
Cost-Effective	[43]
Workflow	[138]
Similarity-Based	[44]
Scope-Based	[143]
Fault-Based	[194], [5], [137], [205], [163], [92], [221], [36], [41]

Table 3.2: Test Case Prioritization Studies Grouped by Technique Type

The methodologies presented in Chapters 5 and 6 can be considered as test case prioritization methods since they entail prioritization: Our cache-based test case scheduling (contribution **C2**) will *prioritize* the next test case that is closer to the one executing in terms of visited basic blocks while our device-based test case scheduling (contribution **C3**) will *prioritize* test cases based on how many tests are executing against each device. However, there are subtle differences when comparing our approaches to traditional test case prioritization techniques:

1. ***Objective -*** In traditional test case prioritisation techniques the objective is to increase the rate of fault detection as well as code coverage. The objective of both our related contributions is to reduce the overall execution time of test suites, an objective that has not been considered before by a test case prioritization method.

2. ***Optimization Timing -*** Traditional test case prioritization methodologies attempt to satisfy a certain criterion **as early as possible** during the execution of a test suite. Once the criterion is satisfied, the order of the remaining test cases is of little significance. In our proposed methodologies, however, the optimization happens *throughout* the execution of the test suite as there is not a cutoff point (e.g. 100% code coverage) in our criterion of choice (i.e. the test suite execution time).

3. ***Scope of Ranking -*** Traditional test case prioritization techniques rank the test cases against the **collection** of test cases executed before them. For example, in coverage-based prioritization a test case will be ranked depending on how much it contributes to the *overall* coverage achieved so far by the collection of test cases executed before it. In our proposed contributions, the *scope of ranking* is narrower as it only takes into consideration the test cases which are executing at the time of ranking and **not** the ones who have finished execution. In contribution *C2* we rank the remaining test cases depending on how similar they are to the test case which is executing at the time or ranking. Similarly, contribution **C3** will rank the remaining test cases depending on how many test cases are executing against each device at the time of ranking.

3.7 Executing Test Cases in Parallel

Once the test cases have been selected and prioritized (see sections 3.5 and 3.6), test case parallelization attempts to execute them in parallel in order to reduce the overall testing time. Parallel test case execution is different from simple task parallelization in that the *same* program is executed in parallel as opposed to the parallel execution of different and independent programs. Executing the same program in parallel might lead to specific system resources being overloaded as well as to certain test dependencies acting as a speedup bottleneck.

- Gupta et. al. propose in [60] a physical (as well as virtual) machine configuration for speeding-up parallel test case execution.

- Haftmann et. al. define in [62] two strategies for executing test cases in parallel on systems powered by databases: The first strategy replicates the database for every test case executing in parallel while the second uses the same database

instance for all concurrent test cases by performing database operation analysis and preventing race conditions.

- Zhengrong Ji et. al. propose in [90] a framework for optimizing the parallel execution of wireless network simulations by reducing the communication, synchronization and scheduling overhead of network operations across multiple processes. Their work is able to speedup significantly the parallel execution of test suites that require simulating network operations.

- The researchers in [51] propose the usage of a functional dependency graph for partitioning a web application test suite into test sets and then distribute these test sets for execution on multiple machines.

- Sasa Misailovic et. al. utilize in [147] the Korat constraint-based algorithm for efficiently generate and execute test cases in parallel. The execution of a test case happens immediately after its generation and a significant speedup is achieved by not storing the test inputs on disk.

- Finally, Sebastian Kappler in [100] models the data dependencies between test cases as a directed acyclic graph (DAG) and incrementally violates these dependencies while checking the test execution result. If the result remains unchanged, the dependency (which is represented by an edge in the DAG) is removed. The result is a test suite with fewer data dependencies between its tests which can benefit more from test case parallelization.

3.8 Scheduling Test Cases

Test case scheduling has received a lot of attention by the research community over the last decades, especially in multi-core system-on-a-chip (SOC) test automation. In [20] the researchers define the test case scheduling problem to be ”the problem of determining start times for the tasks such that the total test application time is minimized”. The authors in [21] show that, from a theoretical perspective, the test case scheduling problem is equivalent to the open-shop scheduling problem which is known to be NP-complete and propose the use of mixed-integer linear programming as a methodology which can produce optimal results for small-scaled systems. In [86] and [22] the researchers present a constraint-driven preemptive test scheduling algorithm which

attempts to minimize SOC testing time while considering resource conflicts and precedence constraints. Finally, in [81] [82] and [226] Yu Huang Et al. formulate the test scheduling problem for core-based SOC as the well-known bin-packing problem and propose a series of heuristic algorithms for achieving acceptable approximations.

The existing literature focuses on test case scheduling on a **single** chip. In heterogeneous application testing, however, we have multiple devices for the test cases to execute against. In Chapter 6 (contribution **C3**), we explore scheduling test cases depending on their target device in an attempt to reduce the execution time of heterogeneous test suites. We empirically evaluate the practical usefulness of a minimal-overhead test scheduling algorithm when applied to a large-scaled industrial test suite.

Chapter 4

Data Transformation for Reducing Test Suite Compilation Time

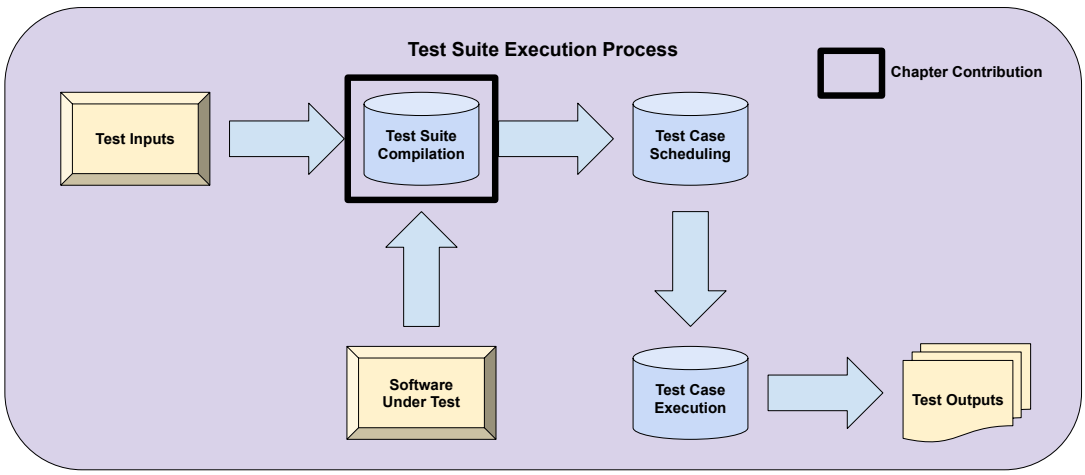

Figure 4.1: Chapter Contribution

Large test suites are difficult to maintain and require frequent compilation. Compiling large pieces of test code is extremely time consuming and severely hinders productivity, as the programmer needs to wait each time he/she wishes to compile and test. Codeplay Software confirm this observation with their testers facing long wait times for test code compilation. In general, it is highly recommended that build or compilation times should be short enough to keep developers focused on the current task, so as to prevent context switching. This is especially important in CI development where tests are compiled and run many times a day, so that even small periods of waiting can add up to significant disruption. This issue is further pronounced in languages like C++ which is known for its long compilation times [110]. However, there has been no

existing work that addresses the problem of prolonged compilation times for large test code. Existing work on compiler optimisation focuses on generating efficient machine level instructions from program source code for fast execution. These optimisations can also be applied to test code for fast execution, but not for *fast compilation*. In fact, these optimisations incur further overhead in compilation times. We target the problem of long compilation times associated with large test codes and aim to achieve significant speedups in compilation, with optimisations that specifically target structure and input data in test code.

In this chapter, we present a novel approach to speedup compilation of test code (see figure 4.1) and empirically assess its benefits. We propose a transformation that restructures *test inputs* and reduces the number of calls to the *software under test* in the test code. The number of instructions in test code reduces significantly with this transformation. We empirically evaluate the effect of the proposed data transformation on test code compilation using two popular C compilers - GCC [55] and Clang [114], enabling all their optimisations. We use industry standard benchmarks – applications from the automotive and telecom domains of the EEMBC benchmark suite [161] for embedded systems, and compute intensive performance benchmarks from SPEC [35]. We also use an industrial application developed by Codeplay Software – ComputeCpp™ [30] which enables acceleration of C++ applications on heterogeneous compute systems using the SYCL [57] open standard. Tests for this application were developed by Codeplay developers as part of test driven development. We evaluate *compilation speedup, execution time, correctness* and *scalability* after applying the proposed data transformation on these benchmarks.

Our approach resulted in significant compilation speedups in the range of 1.3× to 69×. Statistical analysis of the results revealed that our transformation resulted in compilation speedups with both GCC and Clang over all subject programs at 5% significance level. Speeding up the compilation time with the proposed transformation did not negatively impact the execution time of test code. Execution for the Codeplay application is, in fact, faster than the original test code compiled with fully enabled optimisations. We also confirmed that the transformation maintained correctness with respect to results of the test executions, and enabled compilation of large test suites ($>$ 1 million tests) that would otherwise not have been possible.

The rest of this chapter is organized as follows. Section 4.1 presents our approach for reducing compilation time of test code. Our experimental methodology is described in Section 4.2. Section 4.3 presents the results from our experiments. Section 4.4

discusses the threats to validity in our experiment and finally, Section 4.5 concludes.

4.1 Data Transformation

A typical test in a test code, as described in Section 4, comprises of four steps: a set up call, test function invocation with a set of inputs, verification that the outputs match expectations, and a clean up of state and resources used by the test. Listing 4.1 shows a test code sample from Google Test [175], a popular C++ framework for test code development and execution. There are two test groups in listing 4.1, also referred to as parameterized test suites in Google Test. Each contains multiple tests of the respective function under test (FUT) – *IsPrime()* and *GetNextPrime()*. In both groups, each test uses a separate invocation of the FUT over a specific test input, and compares the output to the expected output. Test code in this form has a large number of function invocations and memory operations, which in turn creates significant overhead during compilation. The larger the number of test cases, the longer the compilation time, which in turn has negative impact on productivity.

```
1  TEST_P(PrimeTableTestSmpl7, ReturnsTrueForPrimes) {
2    EXPECT_TRUE(table_->IsPrime(2));
3    EXPECT_TRUE(table_->IsPrime(3));
4    EXPECT_TRUE(table_->IsPrime(5));
5    EXPECT_TRUE(table_->IsPrime(7));
6    EXPECT_TRUE(table_->IsPrime(11));
7    EXPECT_TRUE(table_->IsPrime(131));
8  }
9
10 TEST_P(PrimeTableTestSmpl7, CanGetNextPrime) {
11   EXPECT_EQ(2, table_->GetNextPrime(0));
12   EXPECT_EQ(3, table_->GetNextPrime(2));
13   EXPECT_EQ(5, table_->GetNextPrime(3));
14   EXPECT_EQ(7, table_->GetNextPrime(5));
15   EXPECT_EQ(11, table_->GetNextPrime(7));
16   EXPECT_EQ(131, table_->GetNextPrime(128));
17 }
```

Listing 4.1: Google Test Sample Code

Our approach operates on the *test code*, rather than program source code and transforms the way in which FUTs are invoked and test input data is distributed within test

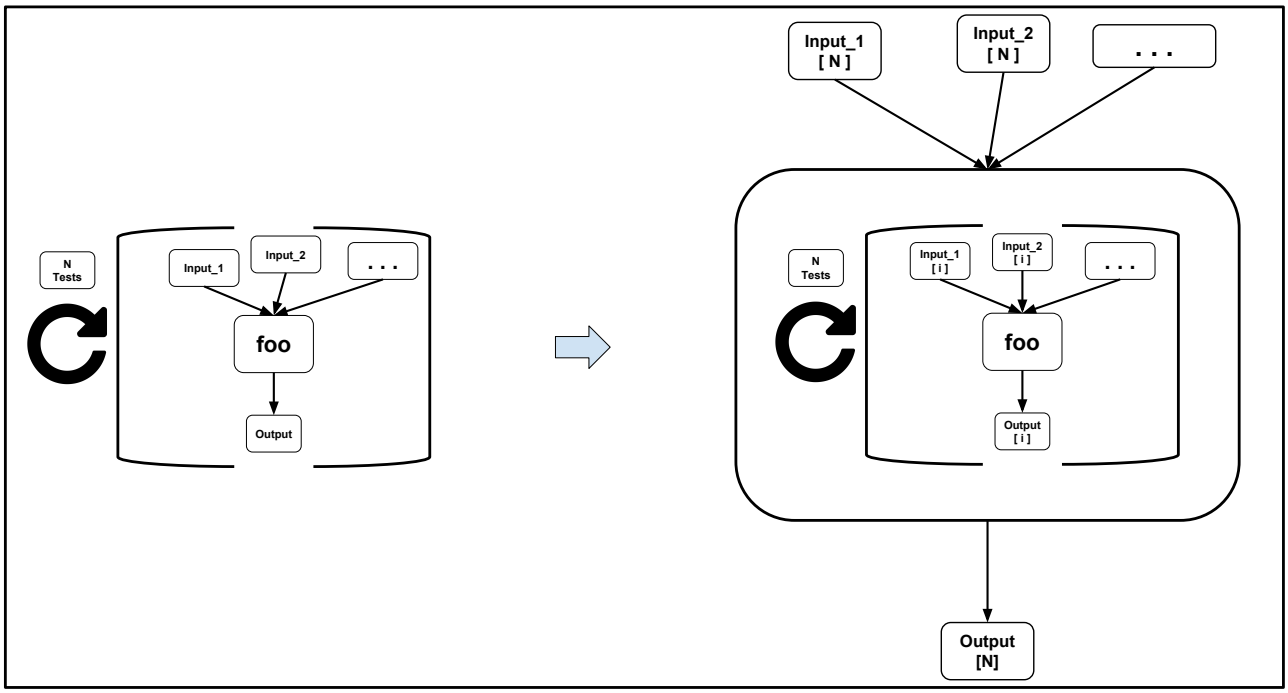

Figure 4.2: Original Test Code with N FUT Calls (left) Transformed to an Equivalent Test Code Containing a Single FUT Call within a Loop (right) Using our Transformation

groups. This is illustrated in Figure 4.2 – we combine test inputs into central data structures and then embed the call to the FUT within a loop in which each iteration represents a single test. This transformation reduces the number of distinct FUT invocations and the number of data structures, on which the compiler operates.

4.1.1 Transformation Algorithm

Algorithm 1 illustrates the steps in our transformation. It takes two inputs – the test code (*TC*) and the name of the program function (*PF*) being tested. Output is the transformed test code (*TTC*). First, the *TC* is searched to identify all calls to the *PF* along with their input arguments. Next, input test data of the same type across the *PF* calls are combined into centralized data structures (*DS*) accessible by every test. In the next step, *DS* are inserted in the *TTC*. Then, the PF calls are updated in the *TTC* to accept the correct data slice from *DS*. The final step combines the PF calls into a single call inside a loop. As part of the final step, for each test, the input data from *DS* is indexed using the appropriate loop iteration number.

Figure 4.3 shows an example of the test code transformation. In the original test code before transformation, there is a separate call to the *foo* function in ev-

ery test. Inputs to *foo* are a one-dimensional integer array, `inputArray[]`, and an integer, `inputScalar`. After the code transformation, the one-dimensional input array passed to each of the tests is replaced by a single two-dimensional array, `inputArray[NUM_TESTS][]`, and the input integer is replaced by a single one-dimensional integer array, `inputScalar[NUM_TESTS]`. Further, multiple calls to the *foo* function are replaced by a single call embedded within a loop, where each iteration represents a test. The iteration index is used to access the correct slice of input data from the merged data structures for each test.

4.1.2 Implementation

The approach is implemented using Python scripts, which take the FUT calls and data structures within the parametrized test suite as inputs. The scripts produce valid C/C++ code in which data structures of the same type are combined into centralised data structures and multiple test function calls are replaced by a single test function call bound within a loop. The scripts also add the index to the correct data slice from the centralised data structure which is passed into the FUT called in each loop iteration.

Input: TC test code, PF program function
Output: TTC transformed test code

1: Create a copy of TC, call it TTC.
2: Search TTC for parameterized test suites, and record all calls of PF, its input arguments and the test oracle data.
3: Merge the input data of the same type from all the tests into centralized data structures ID_DS.
4: Merge the test oracle data of the same type from all the tests into centralized data structures TOD_DS.
5: Merge the multiple PF calls into a single PF call embedded in a loop with as many iterations as there are tests in the parameterized test suite.
6: Update the PF call within the loop so that it accepts the correct slice of data from ID_DS.
7: Update the test oracle assertions within the loop so that they accept the correct slice of data from TOD_DS.
8: Return TTC.

Algorithm 1: Data Transformation

Figure 4.3: Data Transformation Example

4.2 Experiment

We evaluate the effectiveness of the transformation proposed in Section 4.1 using programs from industry standard benchmark families and an industrial application from Codeplay. We seek to investigate the following questions regarding performance and correctness:

Q1. Compilation Speedup: *Does the proposed transformation, relative to existing compiler optimisations, speedup test code compilation?* To answer this question, we used test suites of varying sizes, from 10 to 10K tests, for each subject program and measured the compilation times before and after the transformation, enabling all existing compiler optimisations.

Q2. Execution: *Does the transformation slow down execution of the test code?* To examine this question, for each program and associated test suite, we compare running times of the original and transformed versions of the test code.

Q3. Correctness: *Does the transformation preserve correctness of test executions?* For each benchmark and associated test suite, we compared values of internal states and outputs, obtained during execution of each of the tests in the suite with the original and transformed test code.

Q4. Scalability *Does the transformation enable the compilation of larger test suites?* To answer this question, we evaluated feasibility of compiling the test code with an increasing number of tests, with and without our transformation.

Subject	Domain	Description	#Tests
a2time01	EEMBC - Automotive	Angle-to-time conversion	10K
aifftr01	EEMBC - Automotive	Fast Fourier transforms	10K
aifirf01	EEMBC - Automotive	Finite Impulse Response filter	10K
aiifft01	EEMBC - Automotive	Inverse Fast Fourier transforms	10K
rspeed01	EEMBC - Automotive	Road speed calculation	10K
autcor00	EEMBC - Telecom	Cross correlation of signals	10K
conven00	EEMBC - Telecom	Convolutional encoding	10K
fbital00	EEMBC - Telecom	Bit allocation	10K
fft00	EEMBC - Telecom	Fast Fourier transforms	10K
viterb00	EEMBC - Telecom	Viterbi decoding	10K
401.bzip2	SPEC - Integer	Compression	10K
462.libquantum	SPEC - Integer	Quantum computing	10K
444.namd	SPEC - Floating Point	Molecular dynamics simulation	10K
470.lbm	SPEC - Floating Point	Computational fluid dynamics	10K
999.specrand	SPEC - Floating Point	Pseudo-random number generation	10K
bufferTS	ComputeCpp™	Arithmetic operations on the cl::sycl::buffer class	10K
imageTS	ComputeCpp™	Arithmetic operations on the cl::sycl::image class	10K

Table 4.1: Subject Programs Used in our Experiment

4.2.1 Subject Programs

In this Section, we describe the programs and associated tests used in our experiment. We used 15 subject programs from 2 industry standard benchmark suites, EEMBC and SPEC, that cover a wide range of applications. We also evaluate our approach using an industry provided program, ComputeCpp™, developed at Codeplay Software. Subject programs in EEMBC and SPEC benchmarks were accompanied by a small number of tests. In order to evaluate our approach with large test suites, we randomly generated up to 10K tests for each of the programs in EEMBC and SPEC, using python's *random* library. Tests for ComputeCpp™ were written by developers at Codeplay Software. The programs and their descriptions along with number of tests are provided in Table 4.1.

EEMBC - We used 10 subject programs from the Embedded Microprocessor Benchmark Consortium (EEMBC) [161] that provides a diverse suite of benchmarks organised into categories that span numerous real-world applications. EEMBC benchmarks are *not* just processor-based. They focus heavily on embedded

software running on smartphone, tablets, and other embedded systems. We use 5 benchmarks from the automotive domain (AutoBench) and 5 from the Telecommunications domain (TeleBench) of EEMBC. Benchmarks from AutoBench used in our experiment include a Fast Fourier transformation program, an angle-to-time converter, an inverse Fast Fourier transformation program, a Finite Impulse Response filter and a road speed calculator. The other 5 EEMBC benchmarks come from the telecommunications domain and comprise a convolutional encoder, a bit allocator, a viterbi decoder, a signal correlation program and another Fast Fourier transformer. For each of the 10 EEMBC programs, we randomly generated 10K tests. Test suite sizes of thousands of test cases are not uncommon in embedded software. They typically tend to have more test cases than other forms because of their complexity [38]. Tests for EEMBC programs in our experiment are large input arrays.

SPEC - In addition to the EEMBC benchmarks, we used another 5 benchmarks from the Standard Performance Evaluation Corporation (SPEC) [35] CPU2006 - a benchmark family designed for comparing the performance of different computer systems against compute-intensive workloads. 2 of the SPEC benchmarks, a file compression program and a library for the simulation of a quantum computer, come from the CINT2006 suite which evaluates compute-intensive integer performance. The other 3 benchmarks are part of the CFP2006 suite (compute-intensive floating point performance evaluation) and consist of a biomolecular systems simulator, an incompressible fluids simulator and a pseudo-random number generator. We randomly generated 10K tests for each of the 5 programs.

ComputeCpp™ - We also applied our approach on an industrial application - ComputeCpp™ is Codeplay Software's implementation of the SYCL [57] standard. SYCL is a single-source C++ programming model for OpenCL [185] that provides a high level abstraction over OpenCL, involving data dependency handling and task scheduling. SYCL is comprised of a C++ template library and a device compiler. In order to provide this higher level of abstraction, the features of SYCL involve a very high amount of complexity in their implementation and a combinatorial explosion of potential use cases in their API. ComputeCpp™ enables integration of parallel computing into applications and accelerates code across OpenCL devices such as GPUs. As part of their Test-Driven Development process [11], Codeplay Software has produced a large number of test suites for

ComputeCpp™ with the number of tests in each test suite ranging from hundreds to millions. The compilation time of the test suites for the ComputeCpp™ project has an impact on the software life-cycle because of continuous integration: before each commit gets accepted, all test suites have to be compiled and executed. For ComputeCpp™, the compilation time of its test suites is comparable to their execution time. We applied our approach to two test suites for ComputeCpp™, one for testing the SYCL buffer class and one for testing the SYCL image class. Each test suite contains 10K tests written by Codeplay Software developers.

4.2.2 Measurement

We run our experiments using a desktop computer powered by an Intel Core 2 Duo E8400 processor at 3 GHz, 32KB of Instruction Cache, and 32 KB of L1 Data Cache. The machine runs Ubuntu Server 14.04 with Linux kernel 3.16.0.33. For increased accuracy, we disable any non-critical services on the Ubuntu server while benchmarking. For ComputeCpp™, a desktop computer powered by an Intel Quad Core 6700 processor at 3.4 GHZ with 128KB of Instruction Cache and 128KB of L1 data cache was used. The system also included an AMD Radeon GPU 5450 series with 80 stream processors. We measure compilation and execution time using the Unix *time* command. The results we report consist of the running time on the CPU (*user* statistic). In our experiments, we used two well known C compilers, GCC 7.2.0 [184] and Clang 5.0 [114], for EEMBC and SPEC programs. For ComputeCpp™, the developers use Codeplay's in-house compiler built on Clang. All subject programs were compiled with the highest level of optimisation (*-O3*).

4.3 Results and Analysis

For each of the subject programs presented in Section 4.2, we compare compilation times, execution times and correctness before and after transformation. We collected 10 measurements for compilation and execution times, and report their medians for comparison. We do not report the standard deviation as it was less than 1% for every measurement.

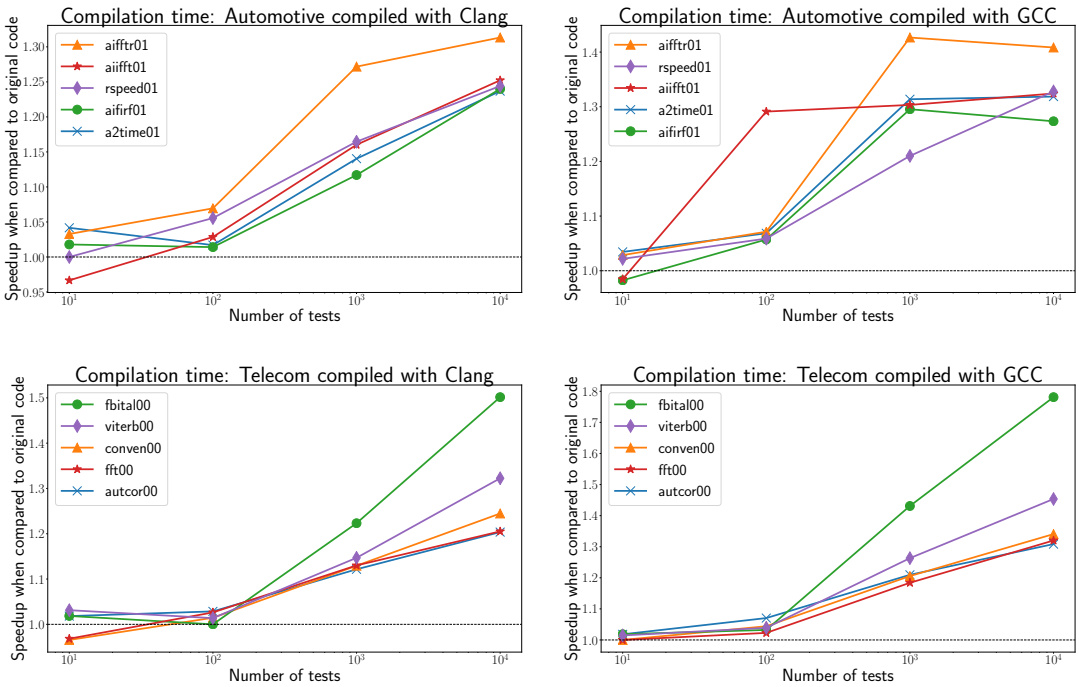

Figure 4.4: Speedup in Compilation Time for EEMBC when Compared to the Original Code for Different Test Suite Sizes

4.3.1 Q1. Compilation

Figures 4.4 and 4.5 show the speedup gained in compilation time with EEMBC and SPEC programs for test suite sizes ranging from 10 to 10K tests, separately for Clang and GCC. Figure 4.6 shows the compilation speedup for two test suites of ComputeCpp™, compiled with Codeplay's in-house compiler. SYCL, being an abstraction layer, allows the host and kernel code of a heterogeneous application to be contained in the same source file. As a result, we present two different plots for compilation speedup: one for the host test code, and the other for the kernel test code. In the following sections, we present speedup results for each of the benchmark families.

4.3.1.1 EEMBC: Automotive and Telecom

The results for EEMBC programs in Figure 4.4 are shown separately for programs from the automotive domain and those from telecom domain to ease illustration. We find that compilation speedup increases with increasing numbers of tests, for all programs in both domains, using both GCC and Clang. Speedup is observed for test suite sizes greater than 100 tests. Maximum speedup for all benchmarks is achieved at the largest test suite size of 10K tests. Original compilation times for 10K tests are of the order of 7 to 10 seconds with Clang, and 10 to 33 seconds with GCC.

For automotive programs, maximum speedup achieved with the Clang compiler is 1.3× for the aifftr01 benchmark (9 secs to 6.5 secs), and 1.4× with GCC for the same benchmark (11secs to 7.7 secs). The average speedup for 10K tests across all benchmarks is 1.3× for both Clang and GCC.

For the telecom benchmarks, maximum speedup achieved with Clang is 1.5×, and 1.8× with GCC for the fbital00 benchmark (6.2 secs to 3.5 secs). The average speedup for 10K tests across all telecom benchmarks is 1.3× for Clang and 1.4× for GCC.

4.3.1.2 SPEC

Figure 4.5 shows the speedups achieved for the SPEC benchmarks. Similar to EEMBC, the speedups are higher for larger test suites and maximum speedup is achieved for 10K tests. We start to observe speedup when number of tests exceeds 100 and the increase is sharp when number of tests rises over 1000. This is because with larger numbers of tests, significantly more number of instructions are reduced with our transformation. This is explained in more detail in Section 4.3.1.5

Original compilation times for SPEC are in the range of 1 to 12 seconds. Unlike

EEMBC, there is a wide range in the maximum speedup achieved over the different programs with both Clang and GCC. With Clang, the maximum speedup achieved is 15× for 470.lbm, but only 1.5× for 401.bzip2. With GCC, the maximum speedup is higher - 20.2× for 999.specrand versus 3.2× for 401.bizp2. Average speedup for 10K tests across all programs is 7.9× for Clang and 12.1× for GCC. High disparity in maximum speedup achieved across programs is due to the number of compilation units associated with each program, and is discussed in depth in Section 4.3.1.5.

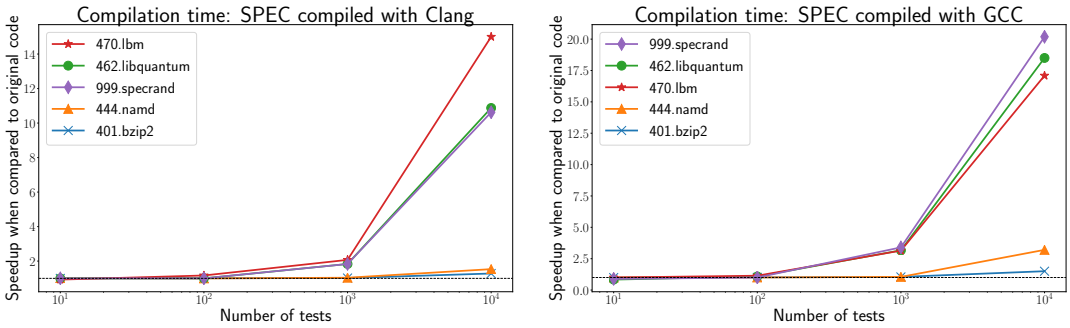

Figure 4.5: Speedup in Compilation Time for SPEC when Compared to the Original Code for Different Test Suite Sizes.

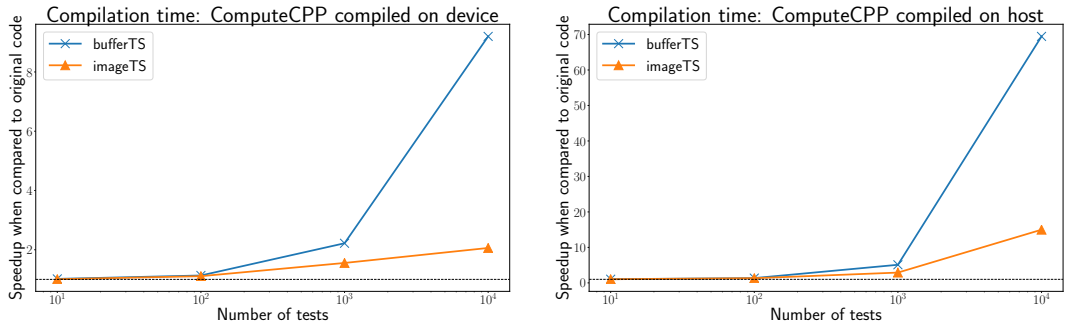

Figure 4.6: Speedup in Compilation Time for ComputeCpp™ when Compared to the Original Code for Different Test Suite Sizes

4.3.1.3 ComputeCpp™

Figure 4.6 shows the compilation speedup achieved for the two ComputeCpp™ test suites – bufferTS and imageTS. Original compilation times are shown in Table 4.2. As observed with EEMBC and SPEC, speedups are proportional to the number of tests being compiled - starts at 100 tests and increases sharply beyond 1000 tests. For device compilation, bufferTS and imageTS start with negligible speedups for 10 tests

and reach a maximum of 9.2× and 2×, respectively, for 10K tests. For host compilation, we observe significantly higher speedups. For 10K tests, bufferTS shows a large speedup of 69.5× while imageTS achieves a speedup of 15×. The average values across both test suites are 42.2× for host compilation and 5.6× for device compilation. The reason for the difference between host and device compilation speedups has to do with the fact that the device code, for both test suites, remains unchanged after the application of our transformation. We discuss this further in next Section 4.3.1.4.

4.3.1.4 Common Trends

Across all benchmarks, we start to observe speedup for test suites that have more than 100 tests. In addition, speedup increases with the size of the test suite. These results indicate that our approach is particularly beneficial for programs with large test suites. Large test suites with thousands of tests are not uncommon, given the rate at which software has been growing in size and complexity. The largest speedup values are achieved for the largest test suite size of 10K tests across all programs, maximum being,

- 1.5X for EEMBC, compiled with Clang
- 1.8X for EEMBC, compiled with GCC
- 15X for SPEC, compiled with Clang
- 20.2X for SPEC, compiled with GCC
- 9.2X for ComputeCpp™, device compilation
- 69.5X for ComputeCpp™, host compilation

4.3.1.4.1 GCC vs Clang For all EEMBC and SPEC benchmarks, there is a difference in the speedup achieved by the Clang and GCC compilers, with GCC achieving better maximum speedup than Clang for EEMBC (1.8× vs 1.5×) and SPEC (20.2× vs 15×) benchmarks. Our experimental data reveals that GCC takes longer to compile the original version of the code, compared to Clang. However, with the transformed version, the differences between the two compilers are much smaller. Differences in compilation time between compilers is not surprising, since they use different algorithms and optimisations. Comparing compilers is not the focus of this paper. It is,

however, worth noting that our transformations achieve faster compilation for **both** compilers, with GCC benefiting more than Clang in our experiments.

4.3.1.5 Analysis

To understand the reason for the speedup observed over all benchmarks, we inspected the output generated by the `-ftime-report` flag in the Clang compiler, which outputs detailed timing data for each compiler pass. It showed that for the largest test suite size, the most time-consuming compiler passes are:

1. **Instruction Selection:** choose machine instructions for each instruction in the intermediate representation.
2. **Function Inlining:** analyse function calls to check if they should be replaced with the body of the function.
3. **Combine Redundant Instructions:** analyse instructions to check if they can be combined into fewer simpler instructions.

The time consumed by the above three passes constitutes an average of 47% of the total time. In comparison, using the transformed test code, the same passes are orders of magnitude faster. This is because the passes operate on fewer instructions using the transformed code, when compared to the original test code.

To confirm this, we inspected the assembly code generated for the transformed and original test code. We observed that in the original version, the compiler emits separate calls to the test function for each test. As more tests are added to the test suite, more function calls are emitted, leading to much longer times for instruction selection and function inlining. In contrast, by embedding the test function call in a loop, as shown in Figure 4.3, the need to compile separate function calls for each test is removed and the number of instructions generated by the compiler is reduced, leading to faster compilation times.

For ComputeCpp™, we observe different speedups for host and device compilations (42.2× vs 5.6× average values). The reason for this speedup is in the structure of the host and kernel code. Both bufferTS and imageTS contain a single kernel, within a host function that is called once for every test in the test suite. Our transformation alters the number of calls to the host function being tested, but *not* the kernel embedded within it. In other words, our transformation only targets host code, *not* device code. Given that the device code remains unchanged, it is surprising that we observe speedup

during device compilation. Upon consulting developers at Codeplay who fully understand ComputeCpp™and its test suites, we learned that the device compiler parses the entire test code (including the host code) to create the AST which is then used to identify the kernel code for further compilation. With our transformation, the size of test code is reduced. As a result, the parser for the device compiler operates on a much smaller total code base, resulting in compilation speedup even when device code remains unchanged.

4.3.1.5.1 Speedup and the Amdahl's Law. The proposed data transformation does not only optimise the three compilation passes mentioned in section 4.3.1.5 (instruction selection, function inlining and redundant instruction combination). We can prove this by applying the Amdahl's law [61] which is formulated as follows:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{4.1}$$

where:

- S_{latency} is the theoretical speedup of the whole task $\longrightarrow$ in our case the theoretical speedup of the test suite compilation.
- s is the speedup of the part of the task that is optimised $\longrightarrow$ in our case the speedup of the compilation passes which are optimised by the data transformation.
- p is the proportion of the execution time that the optimised part originally occupied $\longrightarrow$ in our case the proportion of the total test suite compilation time that the optimised compilation passes occupied before the data transformation.

Let us suppose that our data transformation optimises *only* these three Clang compilation passes:

1. We know that, on average, these three passes constitute the 47% of the test suite compilation time (when compiled with the Clang compiler which applies its optimization passes sequentially [114]) before the data transformation, therefore the p value of the equation 4.1 is 0.47 in this case.
2. Let us now suppose that the speedup achieved in those compilation passes because of the data transformation (s value of the equation 4.1) is so big that their

new execution time is zero. We can model this by taking the limit of the function S_{latency} as s approaches infinity:

$$\lim_{s\to\infty} S_{\text{latency}}(s) = \frac{1}{1-p} \tag{4.2}$$

3. Using the equation 4.2 with $p = 0.47$ (see step 1) we get a theoretical speedup of $1.8868\times$. According to Amdahl's law, if the only compilation passes optimised by the proposed data transformation are the three mentioned in section 4.3.1.5, the maximum speedup we can achieve is $1.8868\times$ even if we assume that the execution time of the optimised compilation passes is zero (see step 2). However, in our Clang experiments, we achieved an average speedup much greater than this threshold ($3.5\times$ across all benchmarks compiled with Clang). By *proof of contradiction*, we proved that the proposed data transformation optimises more compilation passes.

Finally, by using again the equation 4.2 with $S_{\text{latency}}(s) = 3.5$ (average speedup achieved for Clang compilations) we get $p = 0.714$. This means that our data transformation optimises, on average, compilation passes that constitute *at least* 71.4% of the original test suite compilation time.

4.3.1.5.2 Speedup Variation Across Subject Programs. Maximum speedup varies greatly across the benchmarks. With three of the SPEC benchmarks (470.lbm, 462.libquantum and 999.specrand) and ComputeCpp™, our approach achieves significant speedup in the range of $10\times$ to $69\times$. However, with the EEMBC benchmarks and two of the SPEC benchmarks, our approach achieves very low speedup (less than $2\times$). To understand this, we use the data supplied by the `-ftime-report` flag in the Clang compiler, which gives us the time spent compiling each individual file in the benchmark program. This measurement showed us that for each benchmark, our optimisation improves the compilation time of the file with the test code, but it does not affect the compilation time of any other source files used by the program. Thus, when compiling the test code, if the time taken to compile tests is much greater than the time needed to compile libraries and other included files in the test code, then our approach is capable of producing significant speedup.

To better understand this effect, we measured the compilation time for the individual test code files as percentage of the total compilation time for all SPEC and EEMBC programs (for test suites of 10K tests, with Clang). For 3 of the SPEC programs that

gave high speedup–470.lbm, 462.libquantum and 999.specrand– majority of the compilation time ($> 97\%$) is spent on the test code. Closer examination revealed that the test code is a single file that links to external pre-compiled libraries. On the other hand, for the other 2 SPEC programs with low speedup– 444.namd and 401.bzip2– and all EEMBC programs, compiling the test code takes less than a third of the total time. The test code for these applications included several files (up to 10), all of which were compiled together with the test code and take much longer than the test code to compile. Consequently, our transformation speeding up the test code has little effect, only making up a small fraction of the total compilation time. To help gain more speedup for such test codes that include large libraries and other files, we recommend pre-compiling these external files/libraries (as is the case for the other 3 SPEC programs) before applying our transformation.

4.3.1.5.3 Speedup Dip on EEMBC Automotive Programs. On two automotive EEMBC programs (*aifftr01* and *aifirf01*) we record a small dip in compilation speedup from 10^3 to 10^4 test cases. The dip occurs **only** when compiling with the GCC compiler. Furthermore, we observe a similar behaviour (although we do not record a dip) with the *aiifft01* and *a2time01* programs (both belong to the same benchmark family) when compiled **again** with GCC: at 10^2 test cases *aiifft01* shows a $1.3\times$ speedup while the other 4 programs of the same family record a speedup around $1.05\times$. However, as the number of test cases increases, *aiifft01* shows minimal speedup increase with its highest speedup at 10^4 test cases being just over $1.3\times$. Finally, *a2time01* speedup increases linearly until 10^3 test cases but its increase at 10^4 test cases is again minimal.

These trends only occur when compiling the EEMBC programs of the automotive domain with GCC. Even when compiling EEMBC programs of the telecom domain (same benchmark family - different domain) with GCC or even the EEMBC automotive programs with Clang, we do not record any dips or minimal speedup increases as the number of tests cases increases above 10^2. Given all this, we hypothesize that our transformation algorithm might not be as effective for certain *types* of programs compiled with GCC as the number of test cases increases.

4.3.1.6 Statistical Analysis

We analyse the results presented in Figures 4.4, 4.5, 4.6 and determine if the following hypotheses are supported,

H1: Transformed test code, using the GCC compiler, compiles *faster* than the original test code.

H2: Transformed test code, using the Clang compiler, compiles *faster* than the original test code.

We are aware that the number of samples used in our experiment is rather small, and would therefore be unreasonable to fit the data to a theoretical probability distribution. We test the hypotheses by not assuming any particular distribution. To do this, we use the *Mann-Whitney-Wilcoxon test*, a non-parametric test with no distributional assumptions. We use the results for compilation time observed with 10K tests over all subject programs, with and without our transformation. ComputeCpp™compiler, based on Clang, is included in the analysis for results using the Clang compiler.

The p-values using Mann-Whitney-Wilcoxon test were *0.028 for GCC* and *0.036 for Clang* rejecting the corresponding null hypotheses for H1 and H2 at 0.05 significance level. Thus, for the case studies in our experiment, the hypothesis that our transformation results in faster compilation of test code, using both GCC and Clang, is *supported* at 5% statistical significance.

ComputeCpp™ Test Code	**# Tests**	**Compiler**	Orig. time (secs)	New time (secs)
`bufferTS`	10K	host compilation	257	3.7
		device compilation	28.2	3
`imageTS`	10K	host compilation	433.8	29
		device compilation	46.2	22.4

Table 4.2: Compilation Times for ComputeCpp™ Test Codes

4.3.1.6.1 Summary. The speedup gained with our approach depends on the number of tests and also on the proportion of test code size with respect to overall code size being compiled. We find that across all programs in our experiment, the larger the number of tests in the test code, the larger the compilation speedup from our approach. This is mainly attributed to the reduced number of function calls, and as a result, fewer instructions that need to be compiled. For our industrial case study, ComputeCpp™, we observed significant speedups (up to 69X), much larger than the performance benchmarks, EEMBC and SPEC, in our experiment. This is primarily because the industrial case study is much larger than the SPEC and EEMBC programs, and the reduction in function calls has a larger effect on compilation time. This effect is

also observed when comparing SPEC and EEMBC. SPEC programs are larger than EEMBC programs, and we find higher average speedup with our approach for SPEC (12X for GCC) than EEMBC (1.4X). The results in our experiment lead us to believe that the proposed transformation will be particularly valuable for large case studies with large numbers of tests, as is the case for ComputeCpp™.

4.3.2 Q2. Execution

For all subject programs, we measured the running times of the original and transformed versions of the test code, after being compiled in fully optimised mode (-O3 for GCC and Clang) for an increasing number of test cases. We collected 10 measurements per experiment. For all EEMBC and SPEC programs, we find that the execution of the transformed test code is as fast as the original code. Their differences in median and standard deviation was $<0.82\%$. For ComputeCpp™, the transformed test code executed faster than the original version as shown in table 4.3:

ComputeCpp™ Test Code	**# Tests**	Orig. time (secs)	New time (secs)
`bufferTS`	10	0.20	0.20
	100	0.35	0.30
	1000	2.20	1.30
	10000	17.28	12.11
`imageTS`	10	0.30	0.20
	100	0.70	0.50
	1000	5.10	3.80
	10000	55.42	41.07

Table 4.3: Execution Times for ComputeCpp™ Test Codes

For the programs in our experiment, the results categorically show that our transformation does *not* slow down the execution of the test code.

4.3.3 Q3. Correctness

For each subject program, we collected outputs and values of internal variables from executions of each of the tests in the test suites, using both the original and transformed test code. We found that for all subject programs, with 10K tests each, the test outputs and values of internal program states between the two versions of the code are an *exact*

match. We can safely conclude that our framework for transforming test code *preserves correctness of test execution* for all 17 benchmarks and test suites in our experiment.

4.3.4 Q4. Scalability

For each of the EEMBC and SPEC programs, we generated test suites with increasing numbers of tests (powers of 10), and attempted to compile them using the -03 optimisation flag (aggressive optimisation). We did not use the ComputeCpp™ benchmark since we could not generate tests and alter the size of the test suite created by Codeplay developers. We hypothesize that our transformation will make it feasible to compile and optimise much larger test suite sizes than would, otherwise, be possible. When number of tests in the test code reached 1 million, the original version of the test code for all benchmarks, with both Clang and GCC, *crashed* during compilation. However, our transformation allowed test code with more than 10 million tests to be compiled successfully with fully enabled optimisations. This demonstrates that our transformation not only leads to faster compilation of test code, but also makes it feasible to compile very large test suites while enabling all optimisations.

4.4 Discussion

4.4.1 Threats to Validity

We see three threats to the external validity of our experiment based on the selection of programs and choice of test suites:

- We chose programs and test suites in our study that did not include template arguments in the test function call. Our approach is not applicable when tests are parameterised with data that needs to be evaluated at compile time, which is the case for template instantiations. Our transformation causes the input for each test to be evaluated at run-time, using the index of the outer loop responsible for repeatedly calling the test function with different inputs at each iteration. Consequently, in its current form, our transformation is not applicable to test inputs that need to be evaluated at compile time. As a result, our results may only generalize to programs and test suites satisfying this constraint.

- Another threat to external validity has to do with the fact that we chose programs that exhibit common assertion logic across all tests of the test suite. In addition

to test input data, our transformation algorithm operates (in the exact same way) on the test oracle data in order to reduce the overall test code size. For test suites where the assertion logic is different (i.e. different assertions) between tests, the transformation algorithm won't be able to optimise the assertion logic and thus it would be less effective.

- The final threat to external validity relates to the test suites used in our study. We used developer created test suites for ComputeCpp™ and randomly generated test suites that are controlled for test suite size for the EEMBC and SPEC programs. We cannot claim that the test suites we used are necessarily representative of all possible test suites. Additional research is needed to assess the performance of the proposed transformation with different test generation frameworks.

4.4.2 Impact on Developer Feedback

In the transformed test code every iteration of the loop represents a test. This has a negative impact on developer feedback when a test fails during execution since the tests do not have a distinct name or a distinct location in the code base. The tests are evaluated at run time (as opposed to compile time for conventional test suites) and the only thing that separates them is the iteration index. When a test fails during the execution of a transformed test suite, the developer would see messages describing the failed assertions but there will be no correlation to a failed test. Of course, the developer could use debug tools, such as the GDB [183], in order to capture the iteration index and then reverse engineer (or reconstruct) the test by examining the centralized data structures in order to identify the test input and test oracle data that caused the assertion failure(s). However this work requires significant effort and its infeasible for the developer to repeat it every time a test fails.

Some potential approaches for automating the developer feedback loop are:

- **Associate the iteration indexes to the original test names.** The transformation algorithm could extract the test names from the original test suite and create a one-to-one mapping between the iteration indexes and test names. Every time a test fails then the test framework could utilise this mapping to display the actual test name.

- **Display the test input and test oracle data automatically.** Every time a test

fails, the test framework could look-up, using the iteration index, the test input and test oracle data that caused the assertion failure(s) and display them automatically to the developer in a structured way.

4.5 Summary

We have presented a novel approach that allows test code for programs to be compiled efficiently. Our approach restructures the test inputs and reduces the number of calls placed by tests to the function being tested. We evaluated the transformation proposed by our approach using automotive and telecom programs from the EEMBC benchmark suite, programs from the SPEC benchmark suite, and 1 industry provided program and test code, ComputeCpp™. We find that our approach results in compilation speedups of up to 69× for ComputeCpp™, up to 20× for SPEC, and up to 1.8× for EEMBC programs. Variation in speedup is attributed to size of the program, and also proportion of test code over total code size being compiled. Speedups also differed based on the compiler that was used; with gcc benefiting more than clang in our experiments. Further, we found that number of tests being transformed directly affected the speedup. For all subject programs in our experiment, the larger the number of tests, the larger the speedup gained from our approach. We also observed that execution of the test code after transformation is as fast or faster than the original test code. Thus, our transformation for compilation time is not detrimental to execution time. Our experiment results also confirmed that the transformation maintained correctness of test execution results across all subject programs and test suite sizes.

Time consumed for test code compilation is bound to get worse in the future with more complex systems and larger numbers of tests. Our approach provides a safe and efficient means for tackling this problem. In this chapter, we have sampled programs from embedded systems, performance benchmarks, and an industrial application.

Chapter 5

Test Case Scheduling for Improving Instruction Cache Locality

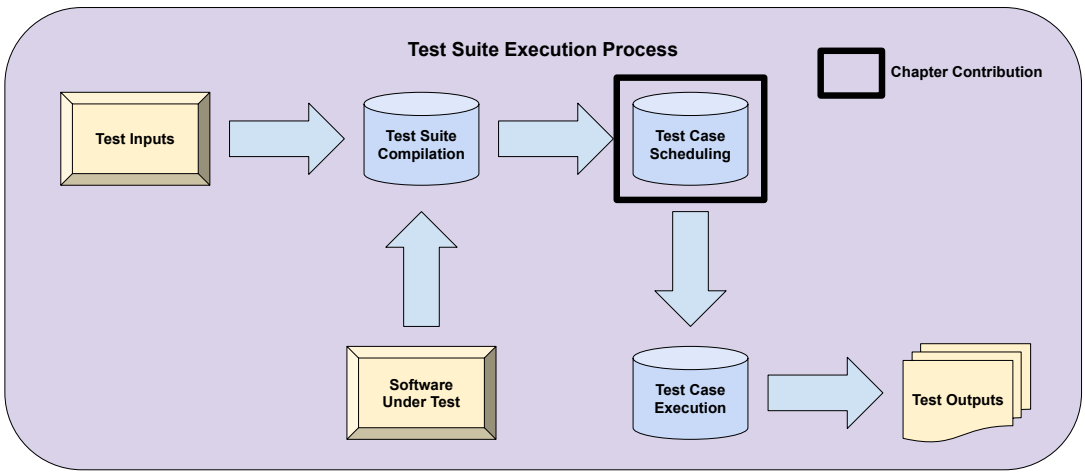

Figure 5.1: Chapter Contribution

The number of test cases needed to effectively test any non-trivial software is extremely large. With the prevalence of software in today's world and the growing complexity of systems, this number is rapidly becoming intractable. Much of the research in software testing over the last few decades has focused on test suite reduction techniques and criteria (such as coverage) that help in identifying the effective test cases to retain. This trend is particularly seen in regression testing and black-box testing where numerous optimization techniques (see section 2.2) have been proposed to reduce testing time. Even after using these test suite optimisation techniques, test suites continue to be very large and their execution is typically very time consuming.

Execution time for present day programs is primarily memory speed rather than

processor speed bounded. Cache misses are a significant consideration for memory speed [199, 203]. It is common knowledge that powerful cache optimizations are crucial to improving the cache behavior and increasing the execution speed of these programs. This observation, however, has surprisingly never been applied to the context of the test suite execution process. In this chapter, we target the test case scheduling step of the test suite execution process (see figure 5.1) and propose a series of test case scheduling algorithms which reduce the number of **instruction** cache misses and, as a result, the execution time of test suites.

Cache misses are reduced by increasing the *locality* of memory references [32], for both data and instructions. Existing literature has only considered improving data/instruction locality over **single** program runs. Enhancing instruction cache locality **across program execution** has previously not been considered and is an entirely novel contribution. The motivation for considering this optimisation is based on the observation that in program testing, we execute the same program several times (albeit with a different *test input*) increasing the chances of encountering repeated instruction sequences. Therefore, the knowledge of common instruction sequences between test cases can be used to help improve the performance of the instruction cache and reduce the overall test suite execution time. Figure 5.2 illustrates the contribution of this chapter for improving instruction cache locality.

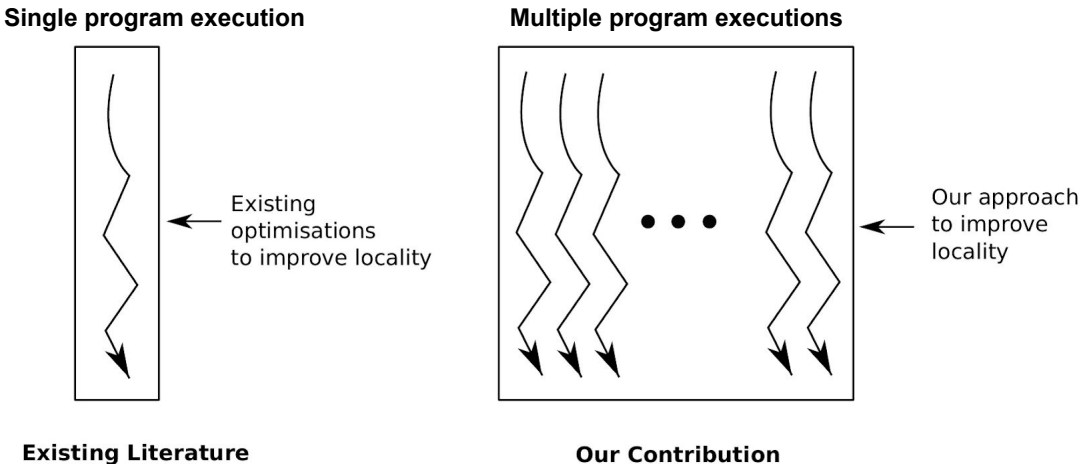

Figure 5.2: Our Contribution vs Existing Work

The chapter is organised as follows. Section 5.1 presents the algorithms and implementations of our approach. Our experimental setup and subject programs are described in Section 5.2. Performance gains with respect to execution time for the different test suite orderings are presented in Section 5.3. Overhead of approximation and

original ordering algorithms is analysed in Section 5.4. We present a further analysis and guideline to using our approach in Section 5.5.

5.1 Instruction-Based Test Case Scheduling

During test suite execution, every test case produces an *instruction trace* that includes the instructions of the software under test which were executed as part of that test case. This *instruction trace* is dynamic in nature and can change as the software under test and/or the test case code evolves. Figure 5.3 includes three test cases for a toy program that compares integers. During *T1* execution the equality condition line 6 executes, succeeds and allows the control flow to enter inside the *if* clause where the *return* statement on line 7 executes and *T1* is completed. The *instruction trace* for *T1* is, therefore, *[6, 7]*. For *T2* the line 6 equality condition fails and the control flow moves to the equality condition on line 9 which is successful and allows the execution of the line 9 *return* statement which completes *T2* with the following *instruction trace*: *[6, 9, 10]*. *T3* produces an *instruction trace* similar to *T2* (*[6, 9, 12]*) with the only difference being that the equality condition on line 9 fails, something which leads to the execution of the *return* statement on line 12.

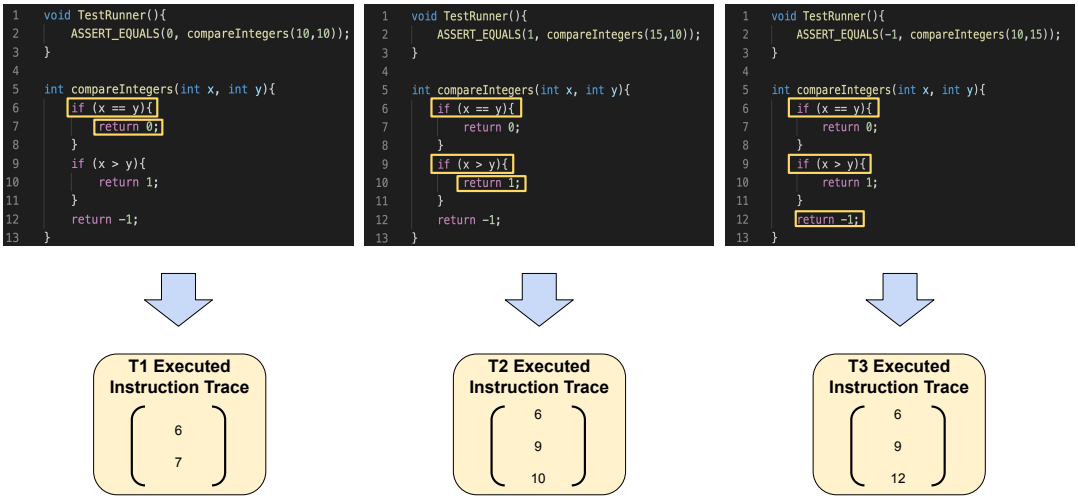

Figure 5.3: Test Case Instruction Traces

To maximise temporal re-use of instructions across several executions of the program (or test case executions), we need to determine an order of test case executions such that distance between consecutive test case executions in the order is minimized while also minimizing the total distance of the order. Note that it is important to min-

imise the total distance additionally, since that helps pick the best among all orders that have minimal distance between consecutive test case executions, each of which is produced by a different starting vertex or test case execution in our case. This is similar to the problem of least cost Hamiltonian Path which is known to be NP-hard. In our approach, we use the nearest neighbour and the approximate nearest neighbour algorithms as approximate solutions since they effectively solve the sub-problem of minimising distance between consecutive test case executions that is important for leveraging immediate temporal locality. Distance between two test cases, T_i and T_j, is defined as:

$$D(T_i, T_j) = \#instructions\ different\ between\ T_i\ and\ T_j \tag{5.1}$$

The rationale for this definition of distance is that instruction locality between test runs is greater when there are more common instructions between them (or fewer different instructions). In the example of figure 5.3, the distance (according to equation 5.1) between *T1* and *T2* is three since there is a single instruction (line 7) which has been executed by *T1* and not *T2* and two instructions (lines 9 and 10) that have been executed by *T2* and not *T1*. *T1* and *T3* have the same distance (three) between them, however *T2* and *T3* have a smaller distance (two) between them since the only instructions executed by one test and not the other are the ones in line 10 (executed by *T2* but not *T3*) and line 12 (executed by *T3* but not *T2*). In such a scenario, we improve the chances of re-visiting the same instructions between two test suite runs if we place *T2* next to *T3*, rather than *T1*, in the order of test case execution.

To enable scalability, we use basic blocks instead of instructions to compute distance in Equation 5.1. $D(T_i, T_j)$ is the symmetric difference between the set of basic blocks visited by T_i and T_j. Note that, $D(T_i, T_j) = D(T_j, T_i)$ in our definition. In our implementation we express the distance as a fraction of the total number of basic blocks visited by all test cases[1], i.e

$$D(T_i, T_j) = \frac{\#basic\text{-}blocks\ different\ between\ T_i\ and\ T_j}{Total\ \#basic\text{-}blocks\ visited\ by\ all\ tests} \tag{5.2}$$

As stated earlier, our approach to solve the distance minimisation problem is based on the nearest neighbour algorithm. For a sequence with N test case runs and T_p being a test case run at position p, our approach re-orders (or permutes) the sequence such that,

$$D(T_i, T_{i+1}) <= D(T_i, T_j),\ where\ j > i+1\ and\ i \in 1, ..., (N-2) \tag{5.3}$$

[1]This is done so that distances can be compared against a threshold defined subsequently.

The condition in Equation 5.3 states that for a test case execution at position i, T_i, the next test case execution in the permuted sequence, T_{i+1}, should be the one that has the least distance to T_i among the test case executions that have not yet been visited (or permuted).

Algorithm 2 illustrates our optimisation approach. We provide as inputs N test cases in some given sequence with T_i being test case at position i. We also provide an input threshold distance, *Thr*, so that test case executions which are within *Thr* distance of each other will be considered neighbours and used in the nearest neighbour computation. Test cases whose distance exceeds *Thr* are not considered neighbours and will be examined for ordering only after all the neighbours are visited. *Thr* is a function of cache size and program size and is used as an indicator of the distance limit beyond which immediate temporal locality between test cases cannot be improved by ordering[2]. This in turn helps save computation effort and time by not having to consider test cases that exceed *Thr* in the nearest neighbour computation.

Steps 1 to 3 of algorithm 2 dynamically analyse test case executions and compute the distance matrix. The heuristic we use to pick the starting test case run in our execution order is the one with most unvisited neighbours. We set this to *current* test case and mark it with a visited flag. We then check if the *current* test case has unvisited neighbours and pick the one that is closest. This becomes the new visited *current* and the process is repeated with neighbours. If there are no unvisited neighbours, and we still have test cases that are not visited, we pick a new *current* test case in the same way as we picked the starting test case in the beginning and repeat the process with neighbours.

5.1.1 Approximate Test Case Scheduling

Algorithm 2 is exponential with respect to number of test cases. The main computational bottleneck is the calculation of the distance matrix which has $\frac{N^2}{2}$ complexity for a test suite with N test cases. We found in our evaluation in Section 5.4 that algorithm 2 is unable to scale beyond 14K test cases.

In order for our approach to be scalable, we implemented an approximate nearest neighbour algorithm which builds a multi-probe locality-sensitive hashing (LSH) index [177] instead of calculating the full distance matrix. LSH is a technique for grouping points in multi-dimensional space into buckets based on some distance met-

[2] *Thr*=1 - (Average #instructions across test runs / Cache size in instructions) if (program size $<$ cache size). Else, *Thr* is median of minimum and maximum distance observed in the distance matrix.

Input: N test cases, P program, Thr defining cutoff distance between test cases to be considered neighbours

Output: List R with the permuted sequence of the N test cases

1: For $1 \leq i \leq N$ run each test case T_i on P and record the set of visited basic blocks for each $\{BT_i\}$ as well as the set of basic blocks visited cumulatively by the test suite $\{BTS\}$.

2: For $1 \leq i \leq N$ combine each $\{BT_i\}$ with $\{BTS\}$ in order to create a binary vector $\{BV_i\}$ of equal length for each test case with each bit representing a basic block.

3: $\forall i, j \in \{1, N\}$ build a $N \times N$ distance matrix of T_i to T_j such that $D(T_i, T_j)$ is the hamming distance of $\{BV_i\}$ and $\{BV_j\}$.

4: From the distance matrix, select a starting test case T as the one that is not visited and has the most unvisited neighbours (i.e. $D(T_i, T_j) < Thr$).

5: Set this to *currentT*, mark it as visited, and insert it into the end of list R.

6: If *currentT* has no unvisited neighbours, go to Step 9.

7: Pick the neighbour that is not visited and has the least distance from *currentT*.

8: Go to step 5.

9: If there are unvisited test case runs in distance matrix go to step 4.

10: Output R as the permuted sequence of test cases.

Algorithm 2: Optimized Order - Nearest Neighbour Analysis

ric (in our case the hamming distance). Points that are close to each other under the chosen metric are mapped to the same bucket with high probability. Our approximation solution is illustrated in algorithm 3. Steps 1 and 2 are identical to our optimisation algorithm but instead of computing the distance matrix, we construct a multi-probe Locality-Sensitive Hashing (LSH) index. We pick a starting test case at random and build an order using approximate nearest neighbour queried from LSH index until the index is empty. The approximation algorithm has O(N) complexity for a test suite with N test cases.

Input: N test cases, P program

Output: List R with the permuted sequence of the N test cases

1: For $1 \leq i \leq N$ run each test case T_i on P and record the set of visited basic blocks for each $\{BT_i\}$ as well as the set of basic blocks visited cumulatively by the test suite $\{BTS\}$.
2: For $1 \leq i \leq N$ combine each $\{BT_i\}$ with $\{BTS\}$ in order to create a binary vector $\{BV_i\}$ of equal length for each test case with each bit representing a basic block.
3: Construct a multi-probe locality-sensitive hashing index $\{LSH\}$ from the set of data points $\{BV_i\}$, $i \in 1,N$.
4: Select a random starting T from the $\{LSH\}$ index.
5: Set this to *currentT*, remove it from $\{LSH\}$, and insert it into the end of list R.
6: If $\{LSH\}$ empty, go to step 9.
7: Query the $\{LSH\}$ in order to get the approximate nearest neighbour of *currentT*.
8: Go to step 5.
9: Output R as the permuted sequence of test cases.

Algorithm 3: Approximated Order - Approximate Nearest Neighbour Analysis

5.1.2 Implementation

We implemented our approach in C++11. Our implementation follows from Algorithms 2, 3 and is illustrated in Figure 5.4.

5.1.2.1 Test Case Analysis

For mapping each test case to the set of its visited basic blocks we used Intel's Pin tool [128]. Pin is an instrumentation-based dynamic analysis framework which allows the development of customized dynamic program analysis tools (a.k.a Pintools). We

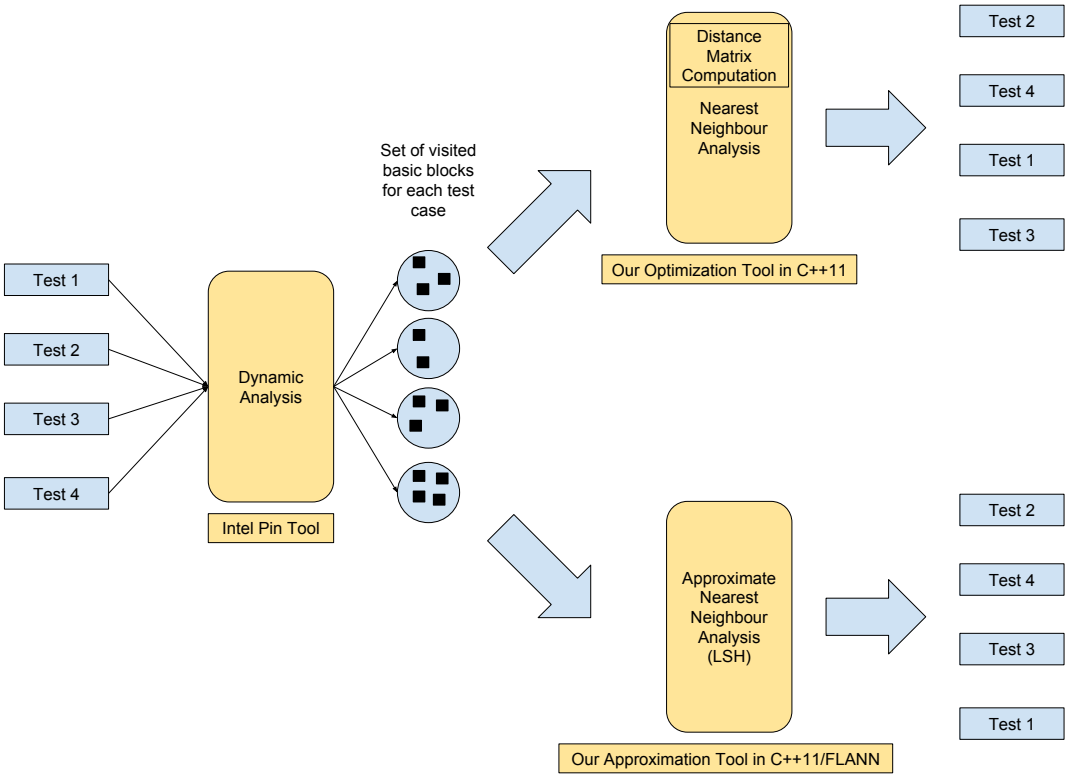

Figure 5.4: Instruction-Based Test Scheduling

developed a Pintool that records visited basic blocks for a program execution. Given a C/C++ program and its corresponding test cases, our implementation will execute each test case independently and dynamically analyse it with our Pintool. To increase the accuracy of our analysis, each subject program is compiled without any optimisations.

5.1.2.2 Test Case Distance Calculation

In our initial implementation, we used the standard C++ library function *std::set_symmetric_difference* for computing the distance between two test cases. However, upon profiling, we found that this function does not scale adequately with respect to the size of visited basic blocks sets. We, therefore, decided to replace the set symmetric difference operation with *hamming distance* between *std::bitsets*, which is semantically equivalent in our context and has been shown to be very fast in C++ [160]. With this improved implementation using *std::bitsets*, the overhead of our original optimisation algorithm was significantly lower.

5.1.2.3 Approximate Nearest Neighbour.

For locality sensitive hashing we used the C++ implementation of FLANN [148], a library for performing fast approximate nearest neighbour searches in high dimensional spaces. In our configurations we had 12 hash tables and the length of the key in these tables was 20.

5.2 Experiment

We conduct our experiments over large sets of programs from different application domains to assess the following,

1. Performance - Execution time of Test Suite. We use four different types of test suites in our evaluation for performance:

- **Opt**- Test suite ordered according to our original optimisation, Algorithm 2 from [188].
- **Approx** - Test suite ordered according to our approximation, Algorithm 3.
- **BC** - Test suite ordered greedily by an existing test adequacy measure. We use branch coverage in our evaluation since it is a widely used structural coverage metric [9].
- **Random** - We randomly permute the test cases in the test suite. We generate 2000 such random permutations. This is done for programs in the SIR benchmark and LLVM Symbolizer. We do not generate random permutations for programs in EEMBC benchmark since the size of test suites are large, 70K tests. Execution time for large number of random permutations becomes impractical for such large test suites.

We also check if the number of *cache misses* have reduced as a result of our optimisations.

2. Overhead of Optimisation. We assess the overhead for computing the optimised and approximated permutations with respect to increasing number of test cases in the test suite.

5.2.1 Subject Programs

We assess performance and overhead over the following programs:

SIR - We use 11 programs from the SIR repository for our experiment. Programs include lexical analysers, priority schedulers, a search utility, stream text edi-

tor, a statistics program, and an aircraft collision avoidance system. Most SIR programs are accompanied by 100 to 5500 test cases. Space is the only subject program in SIR with a moderately large test suite - 13585 test cases. We ran the existing test suite associated with each of the SIR programs for our experiment.

EEMBC - In addition to the SIR repository, we used the Embedded Microprocessor Benchmark Consortium (EEMBC) that provides a diverse suite of benchmarks for microprocessors, micro-controllers and embedded devices. We use 8 EEMBC benchmarks – 3 from the automotive domain (AutoBench) and 5 from the Telecommunications domain (TeleBench) of EEMBC. AutoBench is a benchmark collection for evaluating the performance of microprocessors and microcontrollers in automotive applications and programs used in our experiment include an angle-to-time converter, a pulse-width modulator and a road speed calculator. The other 5 EEMBC benchmarks come from the telecommunications domain consist of a convolutional encoder, a bit allocator, a viterbi decoder, a signal correlation program and another Fast Fourier transformer. For each of the 8 EEMBC programs, we randomly generated 70000 test cases.

LLVM Symbolizer - Finally, we conducted our experiment on an LLVM tool, the `llvm-symbolizer` which takes as input arbitrary object files along with addresses and returns the corresponding source code locations. This tool utilizes debug info sessions and the symbol table of the input object file. We generated 432 test cases which are a combination of object files from well known programs (including SIR and EEMBC) along with a set of randomly generated addresses.

5.2.2 Measurement

We run our experiments using a desktop computer powered by an Intel Core 2 Duo E8400 processor at 3 GHz, 32KB of Instruction Cache, and 32 KB of L1 Data Cache. The machine runs Ubuntu Server 14.04 with Linux kernel 3.16.0.33. For increased accuracy, we disable any non-critical services on the Ubuntu server while benchmarking. We measure the execution time of our algorithms and program test case runs using the Unix *time* command. The results we report consist of the time the under-profiling program was running on the CPU (*user* statistic).

Subject	Description	Size (Avg. Exec. Instrucs.)	#Test Cases	Repository
concordance	Utility for word indicies	3.6e+06	744	SIR
grep	Search utility	2.57e+06	470	SIR
printtokens	Lexical analyser	6.27e+03	4130	SIR
printtokens2	Lexical analyser	9.08e+03	4115	SIR
replace	Pattern matching and substitution	1.28e+04	5542	SIR
schedule	Priority scheduler	5.64e+03	2642	SIR
schedule2	Priority scheduler	1.49e+04	2710	SIR
sed	Stream text editor	5.36e+06	358	SIR
space	Interpreter for ADL	6.16e+04	13585	SIR
tcas	Aircraft collision avoidance system	2.23e+02	1608	SIR
totinfo	Statistics computation	1.89e+04	1052	SIR
autcor00	Cross correlation of signals	6.25e+04	70000	EEMBC
conven00	Convolutional encoding	5.99e+04	70000	EEMBC
fft00	Fast Fourier transforms	2.97e+05	70000	EEMBC
fbital00	Bit allocation	7.74e+04	70000	EEMBC
viterb00	Viterbi decoding	5.19e+05	70000	EEMBC
a2time01	Angle-to-time conversion	5.92e+03	70000	EEMBC
puwmod01	Pulse-width modulation	1.33e+06	70000	EEMBC
rspeed01	Road speed calculation	2.18e+07	70000	EEMBC
llvm-symbolizer	Address to source code conversion	1.70e+06	432	LLVM

Table 5.1: Subject Programs Used in our Experiment

5.3 Performance Results

For each of the different benchmarks – SIR, EEMBC and LLVM Symbolizer, we report the **performance** of the different test suites mentioned in Section 5.2 – `Opt`, `Approx`, and `BC`. For SIR programs and the LLVM Symbolizer, owing to the smaller size of their test suites, we also report the performance of 2000 random permutations of test suites (`Random`).

5.3.1 SIR

Comparison of the four different types of test suites – `Opt, Approx, BC`, and `Random`, for the 11 programs in the SIR benchmark is shown in Table 5.2. The histogram frequencies for the 2000 random permutations, and 100 runs of each of `Opt, Approx` and `BC` are shown. The vertical dashed line shows the median execution time over the distribution for each of the four different types of test suite. We do **not** show standard deviation, since we found that the execution times for all subject programs over the

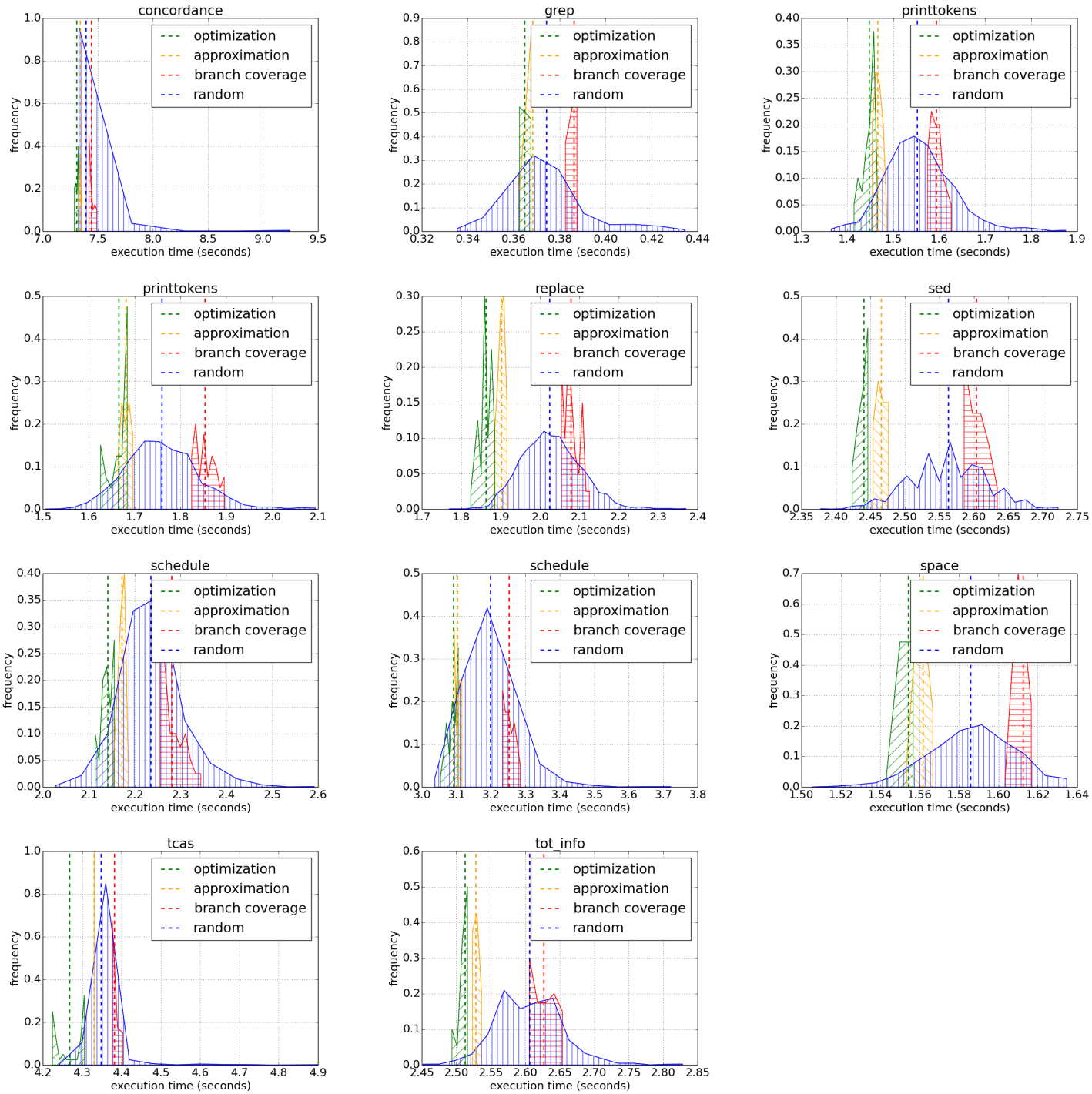

Table 5.2: Histogram Frequencies of Execution Time for `Opt, Approx, BC, Random` Test Suites for 11 SIR Programs

random test permutations were *not normally distributed*. We confirmed this by running chi-squared goodness of fit test, and the p-values for all programs were 0 (rejecting the null hypothesis that they are normally distributed at 5% significance level).

5.3.1.1 Observations on `Random`

It can be seen from the plots in Tables 5.2 that execution times clearly vary across random test permutations. The extent to which execution times vary is different for each program and associated random tests. The differences between the best and worst permutation execution times ranged from 8.53% to 29.38%. The differences observed over random permutations can be attributed to test case distances being distributed over a wide range for these programs. Test suites for these programs were such that there were clusters of test cases with low distances between them, i.e. they execute similar control flow paths. Distances between test cases across clusters were high. As a result, random permutations that change the ordering of test cases within a cluster will have little effect on the instruction locality. and those that changed the order across clusters will have a negative effect on instruction locality. The size and number of clusters will determine the magnitude of this effect.

5.3.1.2 Comparison Across Test Suites

It is evident that for all programs, median performance of `Opt` does better than the majority of the random permutations and is very close to the best performing random permutation (left extreme of the blue curve). For 8 of the 11 programs, `Opt` does better than 90% of the random permutations. For 2 other programs, `Opt` outperforms 83% of the random permutations. As observed earlier, test suites for these programs have clusters of test cases with low distances within, and high distances across clusters. Our approach for permutation ensures that test cases within clusters are executed in close succession, effectively leveraging the instruction locality between them. We believe this is the primary reason for outperforming a large majority of random permutations. We find that median `Opt` outperformed the median `Random` performance by 1.13% to 7.99%. `Approx` also performs comparably to `Opt` on all programs, differences between their medians is between 0.48% to 2.10%. Among the test suite orderings, `BC` typically tends to be worst performing, achieving lesser than the median `Random` performance across all SIR programs. We believe this can be attributed to the insensitivity of the `BC` ordering to instruction similarity between test cases and as a result, incurs an increased

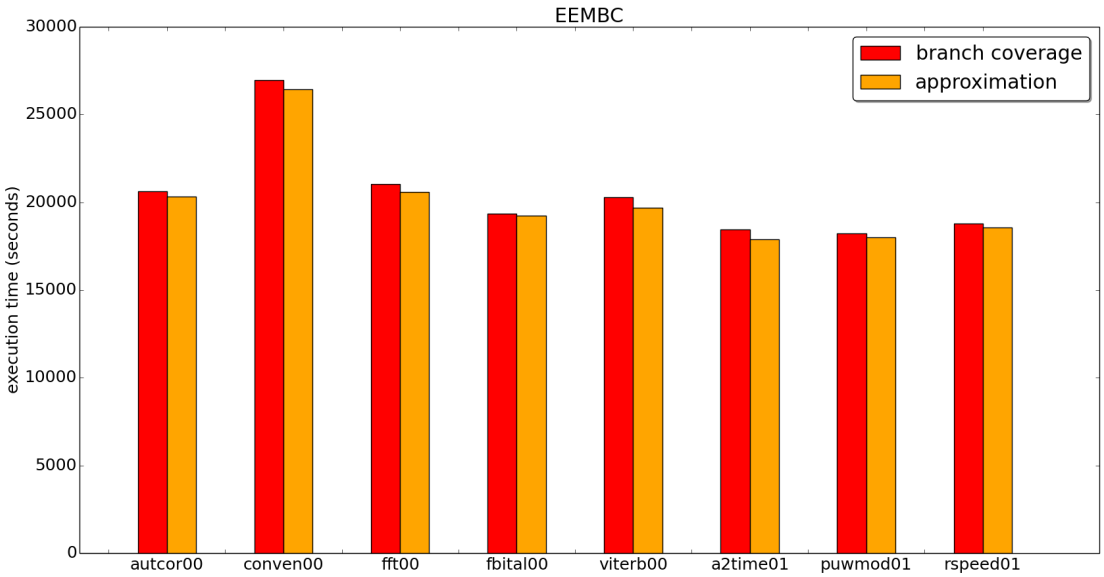

Figure 5.5: Comparison of Execution Times for `Approx, BC` for 8 EEMBC Programs

number of instruction cache misses. We confirm this with the results from cache misses discussed in Section 5.3.4.

For `concordance`, although median `BC` is worst performing, all 4 different test suite orderings are very close in the performance achieved. This is because number of instructions executed per test case for `concordance` exceeds cache size. As a result, it is not possible to achieve improved instruction locality across test cases with our approach for `concordance`.

5.3.2 EEMBC

Each of the subject programs in the EEMBC benchmark were accompanied by 70K randomly generated test cases. Each test suite execution took more than 6.5 hours to execute. In the interest of keeping execution times practical, we did not run 2000 random permutations (`Random`) of each test suite. Additionally, we find that the overhead incurred with our naïve approach to generate the `Opt` ordering was prohibitive for test suites with 70K test cases. Overhead of our algorithms is discussed in Section 5.4. As a result, we restrict the performance discussion for EEMBC programs (with 70K tests each) to comparison of orderings generated by `Approx` and `BC`, as shown in Figure 5.5.

5.3.2.1 `Approx` versus `BC`

As can be seen in Figure 5.5, `Approx` and `BC` are comparable in performance, with `Approx` only slightly outperforming `BC`. Differences in performance are in the range of 0.50% to 2.94% across all 8 EEMBC programs. This may seem surprising after the results observed over SIR programs. However, on further investigation, we find that these results are to be expected. The difference in executed instructions between test cases in the suite is negligible over all programs, implying similar visited basic blocks. We measured the distances among test cases in the suite to confirm this and we found the average test case distance (as a percentage of the total number of executed instructions by the full test suite) was in the range of 0.84% to 3.35% across EEMBC programs. In comparison, average test case distance for SIR programs was in the range of 7.97% to 38.62% As a result, test case ordering will have no meaningful effect on instruction locality, and therefore execution time.

5.3.3 LLVM Symbolizer

For LLVM Symbolizer, as with SIR, we generated all four different types of test suites – `Opt, Approx, BC, Random`. Figure 5.6 depicts the histogram frequencies for 2000 `Random` permutations and 100 runs of each of `Opt, Approx` and `BC`. LLVM Symbolizer showed significant improvements in execution time with both `Opt` (16.74%) and `Approx` (13.06%) relative to `BC`. Furthermore, `Opt` outperformed 97.75% of the `Random` permutations, while `Approx` outperformed 87.6%. Median `Opt` performance was better than median `Approx` by 4.23%.

Performance gains observed over LLVM Symbolizer is highest across all benchmarks in our experiment. The superior gains was a result of high test case distance between tests in the test suite. This is further discussed in Section 5.5.

5.3.4 Conformance with Cache Miss Rate

The premise in our orderings (`Opt` and `Approx`) is that they will reduce the number of instruction cache misses by increasing cache locality. This in turn will translate to faster, or reduced, execution time. We checked this premise for both `Opt` and `Approx` orderings. Cache miss rate was measured by running *Cachegrind* that is part of Valgrind [153] on the subject programs. We find that the reduction in execution times closely follows reduction in cache misses, for our orderings, relative to `BC`, over the

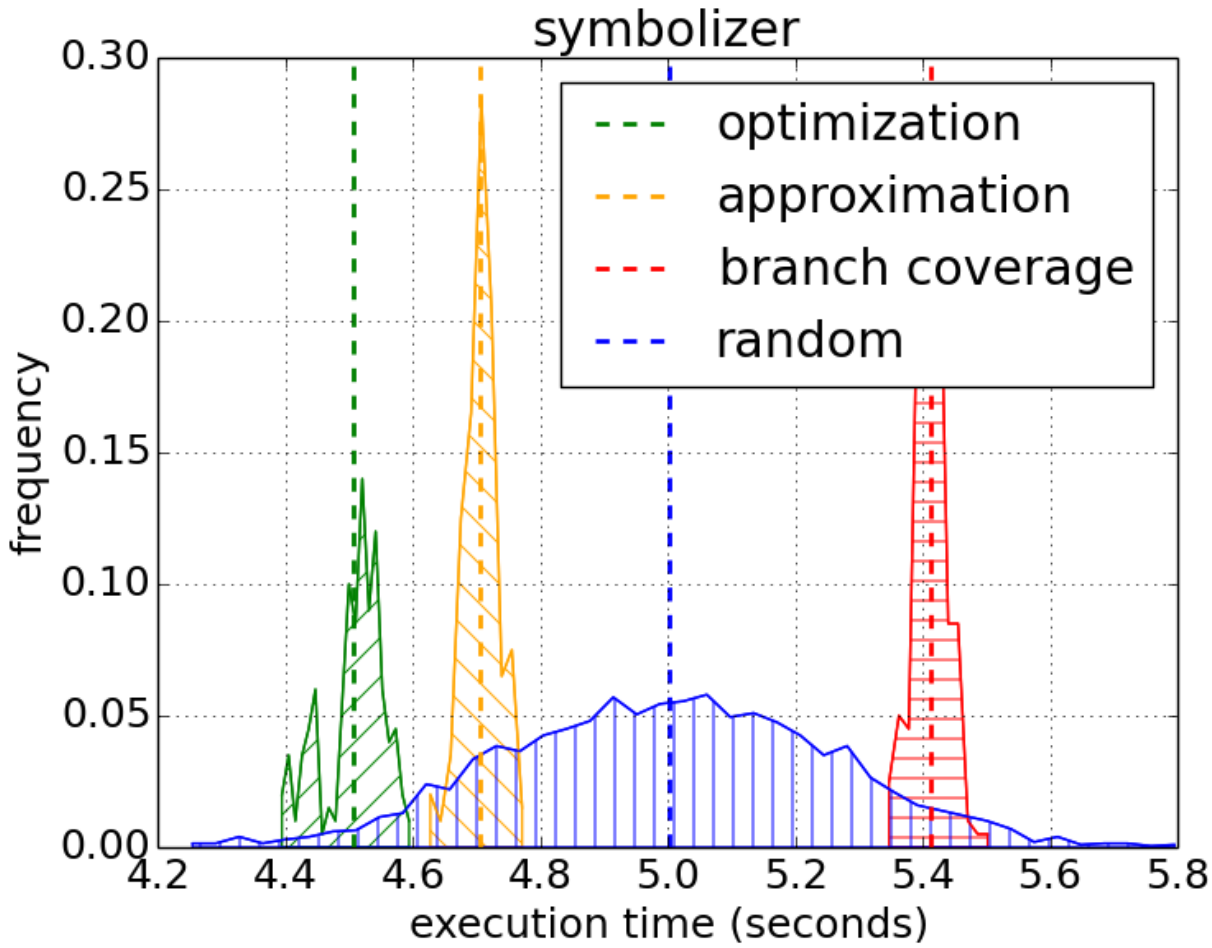

Figure 5.6: Histogram Frequencies of Execution Time for `Opt`, `Approx`, `BC`, `Random` Test Suites for LLVM Symbolizer

subject programs. For instance, for `replace` in the SIR benchmark, with the `Opt` ordering, cache miss rate reduction was 15.98% compared to BC, and execution time was faster by 10.39%. With the `Approx` ordering, cache miss rate reduced by 11.75% with respect to `BC`, and execution time was 8.47% faster. For `fbital00` in EEMBC, cache miss rate reduced by 0.9% and execution time by 0.5% using the `Approx` ordering, when compared to `BC`. `LLVM Symbolizer` acheived 14.84% and 12.03% reduction in cache misses, with `Opt` and `Approx` orderings respectively. Corresponding speedup in execution was 16.74% and 13.06% with both orderings.

5.3.5 Synopsis

It is clear from the performance results for programs in SIR, EEMBC and LLVM Symboliser that the order in which test cases are executed **affects** execution time. The nature of programs and test cases, in terms of range of distances between test cases, determine the magnitude of the effect. The differences between worst and best random permutation ranged between 8.53% to 29.48% over SIR programs and 27.12% over LLVM symbolizer. `Approx` ordering gave comparable performance to `Opt` ordering over all benchmarks. Maximum performance gains were observed with LLVM Symbolizer and least with programs in EEMBC benchmark. Average test case distance between tests is high for LLVM Symbolizer. EEMBC program executions are compute-intensive with limited variation in control-flow (largely sequential). As a result, distance between test cases and scope for improvement is low. We confirmed that our orderings, `Opt` and `Approx`, reduced instruction cache miss rates and the magnitude of the gains followed execution time improvements. `Opt` ordering gives the best performance over random ordering and `BC`. However, it does not scale to large test suite sizes as discussed in Section 5.4.

5.4 Overhead Results

In this Section, we discuss overhead incurred in executing our algorithms for `Opt` and `Approx` orderings. For `Opt` ordering, we use the efficient implementation using *std::bitsets* mentioned in Section 5.1, rather than the implementation in [188]. We compare overhead (time taken) for the two orderings relative to increasing number of test cases until maximum test suite size is reached.

5.4.1 SIR

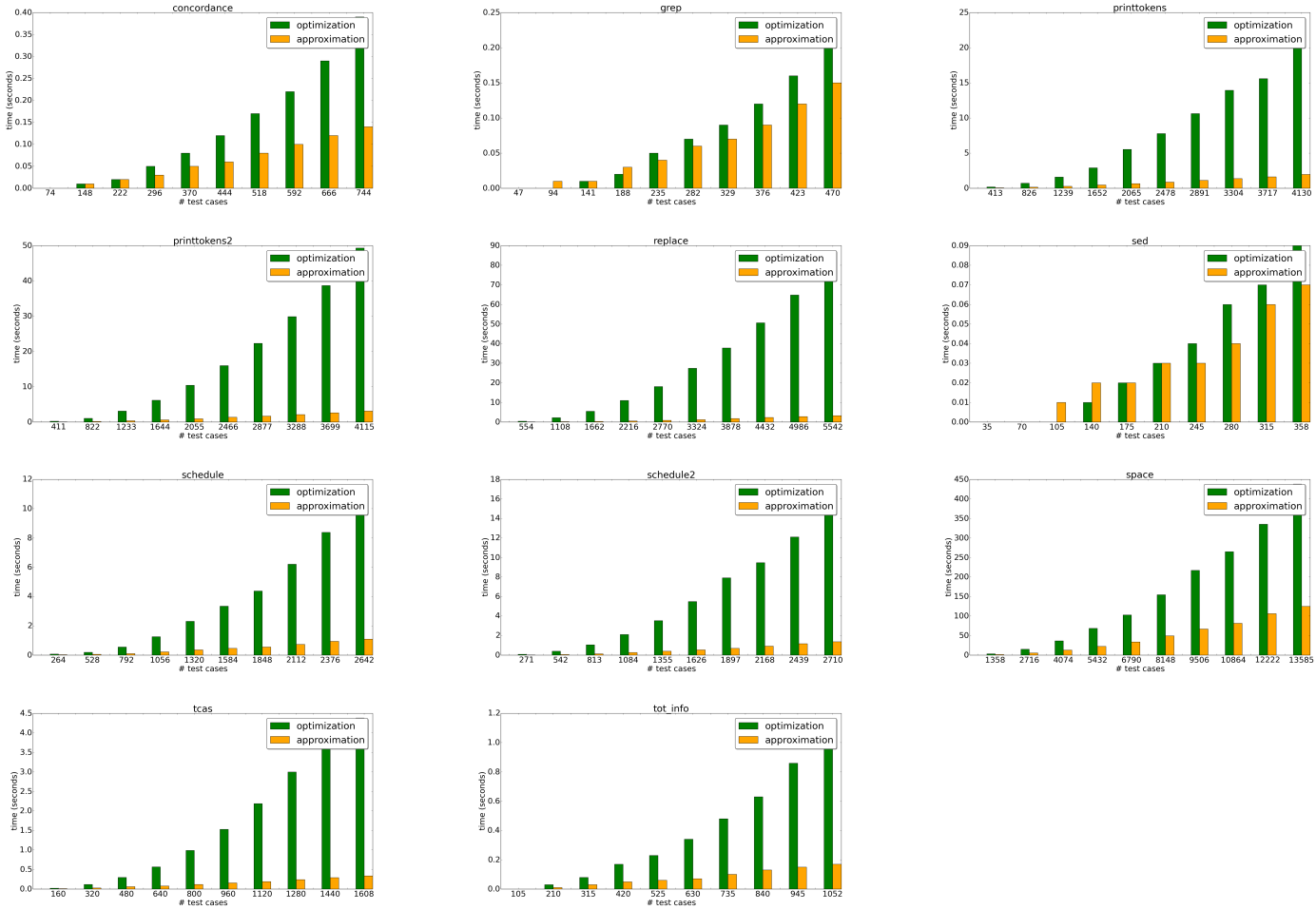

Table 5.3: Overhead for Generating `Opt`, `Approx` Orderings for Increasing Number of Tests over 11 SIR Programs

It is clear from Table 5.3 that overhead of our algorithm to generate `Opt` ordering increases exponentially with the number of tests over all SIR programs. We limited our analysis to the maximum number of test cases available in the repository for each program. For the `replace` program, for instance, overhead for `Opt` starts at 0.44 seconds for 554 tests and increases exponentially to 84.55 seconds for 10 times more tests. Overhead of `Approx` on the other hand, increases linearly with the number of tests. This is observed uniformly over all SIR programs. For `replace`, `Approx` overhead starts at 0.07 seconds for 554 tests and increases linearly to 3.30 seconds for 10 times more tests. Comparing the overhead of the two orderings we find overhead of `Opt` is significantly larger than that of `Approx` when the number of tests is large. Considering the full test suite for all programs, `Opt` overhead is greater than `Approx` overhead by

28% (for sed) to 2462% (for replace). For subject programs, grep, sed, absolute value of Opt overhead is very small (< 0.25 secs) since the maximum number of test cases executed with these programs is small (less than 500). At such sizes, Opt overhead is 28% (sed) and 53% (grep) more than Approx, which is better than for other programs with larger numbers of tests.

5.4.1.1 Smaller Overhead with Approx

Using Approx, rather than Opt, results in considerable overhead reductions across all programs. Overhead of executing the Approx algorithm when compared to test suite execution time is negligible for 4 of the 10 SIR programs with small test suite sizes. Overhead is 0.4 to 1.8 times test suite execution time for all other programs, except space where it is 7 times as much. space has the largest test suite with 13.5K tests, however the average executed instructions and total test suite execution time is small. As a result, the impact of Approx overhead is more significant. Overhead with large test suites for programs and test suites with significantly longer execution times is discussed in the following Section.

5.4.2 EEMBC

Overhead of Opt and Approx for EEMBC benchmarks is shown in Table 5.4 for an increasing number of tests cases, upto 70K tests. We found that the Opt algorithm does not scale beyond 14K tests (runs out of memory). The Approx algorithm, on the other hand, does scale to the maximum test suite size of 70K tests, for all 10 programs. The average Approx overhead, across programs, as a fraction of the total execution time for 70K tests is 22.6%. Overhead of our ordering algorithm can be further reduced by running it on GPUs. We found a reduction of over 5 times in the overhead when running the Approx algorithm for autcor00 with 70K tests on a NVIDIA GeForce GTX 660M with 384 CUDA cores.

5.4.3 LLVM Symbolizer

The overhead of the LLVM Symbolizer is illustrated in Figure 5.7. For Opt, the overhead ranged from 0.0004 seconds (43 test cases) to 0.17 seconds (432 test cases) which represents 3.7% of the full test suite execution time. For Approx, overhead was in the range of 0.001 seconds to 0.28 seconds (5.8% of execution time). Overhead for Approx

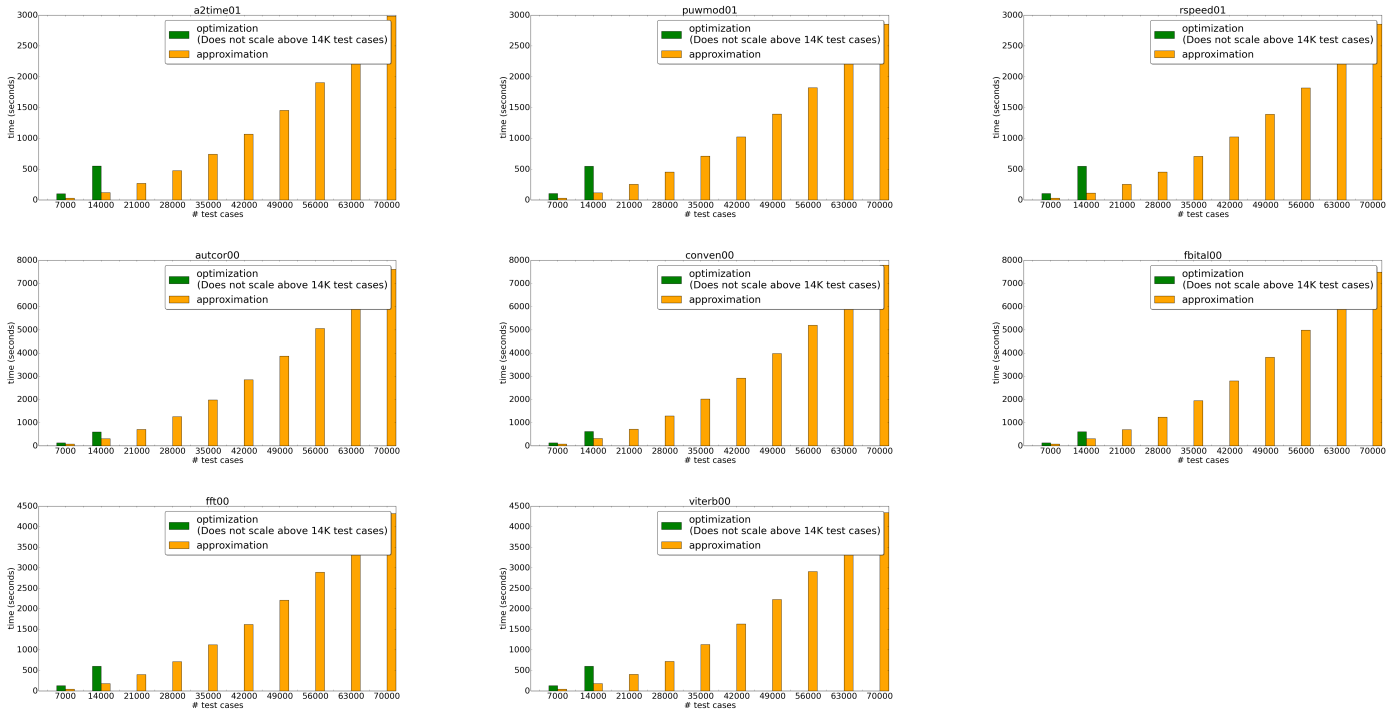

Table 5.4: Overhead for Generating `Opt`, `Approx` Orderings for Increasing Number of Tests over 8 EEMBC Programs

is higher than `Opt` when the number of test cases is small. The time taken to build LSH index in `Approx` is more significant with small test suites. It is, however, worth noting that overhead of `Approx` increases at a slower rate than `Opt` as test suite sizes get larger (owing to lower algorithmic complexity). `Opt` overhead can become prohibitive for large test suites, as observed over EEMBC programs.

5.4.3.1 Overhead - Offline and Amortised

The ordering algorithms, whether it be `Approx` or `Opt`, can be performed offline (before a test suite is deployed), avoiding costly overhead during test execution phase. Additionally, the ordering, once generated, can be re-used for future test suite runs. It is common practice in embedded devices to periodically run in-situ test suites and this is further emphasized by practices like test-driven development [38]. For newer releases on evolving software, overhead can be amortised by executing the ordering algorithm only on new test cases and existing test cases affected by updates. To identify the existing test cases that need to be re-ordered, we will use the information on modified code and determine the test cases that execute modified basic blocks and those reachable from them. This overhead incurred for evolving software is similar to the

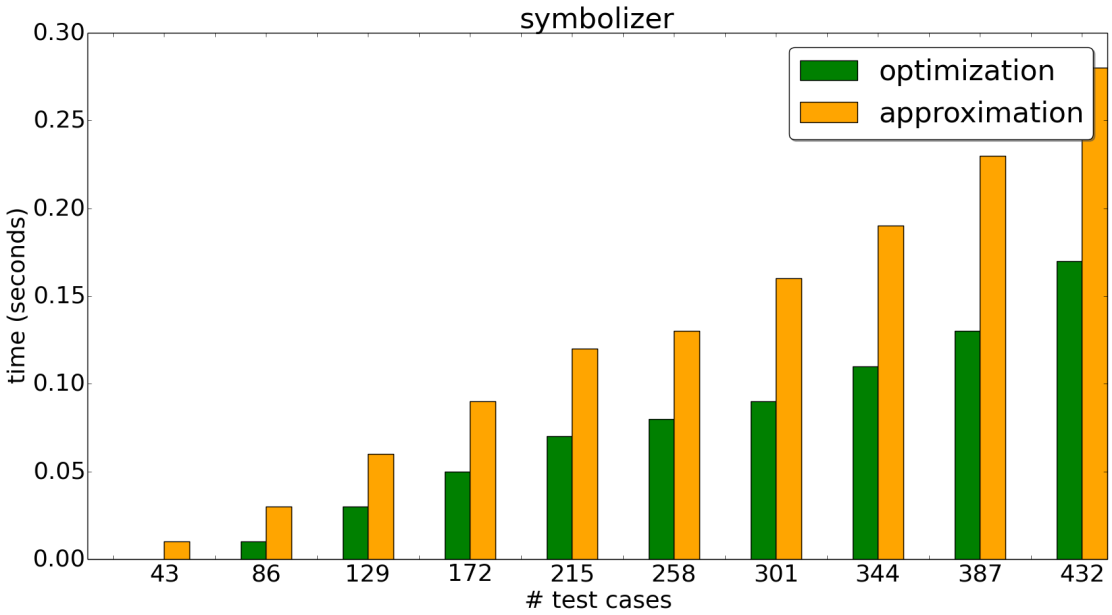

Figure 5.7: Overhead for Generating `Opt`, `Approx` Orderings for Increasing Number of Tests for LLVM Symbolizer

overhead incurred in regression test selection or prioritisation techniques.

5.4.4 Synopsis

The overhead of `Opt` is exponential in size of test suite and does not scale beyond 14K tests for EEMBC programs. When test suite sizes are small ($<$ 500 tests), overhead of `Opt` is acceptably small as seen with LLVM Symbolizer. The overhead of `Approx` is tractable and scales well to large test suites (70K tests for EEMBC). We found that the overhead could be further reduced with the use of GPUs. Additionally, ordering algorithms can be performed offline and overhead need not be incurred during actual test suite execution.

5.5 Discussion

Our results in Sections 5.3 and 5.4 indicate that `Opt` and `Approx` orderings targeting cache locality result in faster execution times over a conventional ordering like `BC`, but with varying magnitudes for the different subject programs. In this Section, we analyse reasons for this and present a metric that can be used to help predict gains from our approach and to make informed decisions on whether or not to apply our proposed

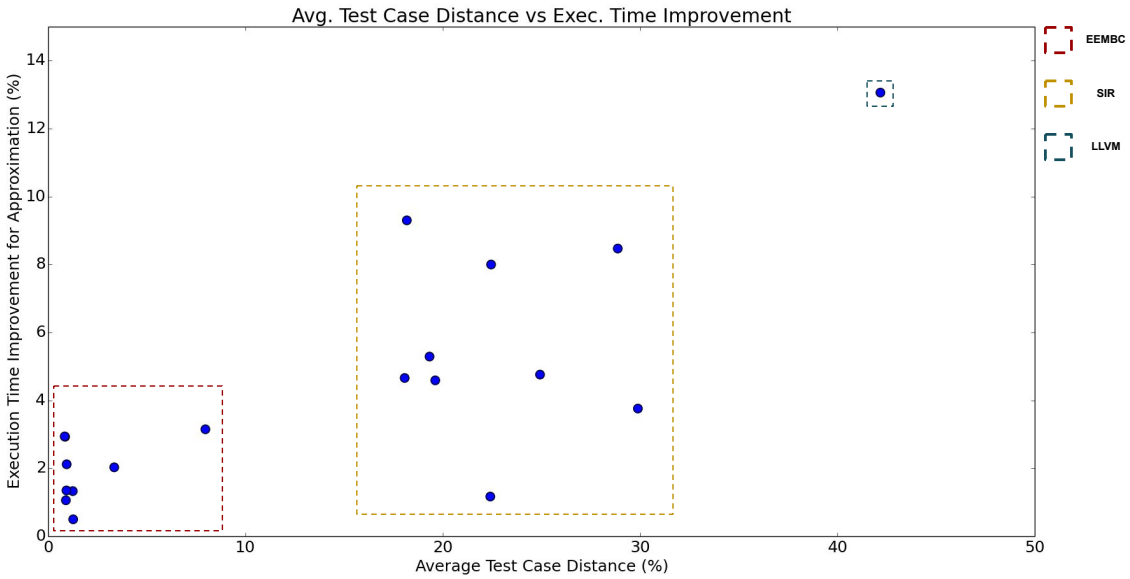

Figure 5.8: Test Case Distance vs Time Improvement for Approx Ordering over BC

orderings.

Figure 5.8 shows average Test Case Distance (TCD), and execution time improvements of `Approx` over `BC` with full test suites for each subject program. Recall that `Opt` ordering does not scale for EEMBC programs with large test suites, so we only analyse the results for `Approx`. TCD between two test cases is computed as a fraction using Equation 5.2, number of different basic blocks between the two tests over total executed basic blocks by a test suite. For a test suite of size N, the distance matrix between test cases is a square matrix (T_{ij} is symmetric to T_{ji}) with diagonal entries being 0. The number of test case distances that are computed in such a matrix is $N^2/2 - N$. For each subject program, we average over all such test case distances to get the `Y-axis` in Figure 5.8.

We find that average TCD is positively correlated with `Approx` execution time improvement, $r = 0.76$. We do not include the `concordance` program in this computation since a single test execution exceeds cache size and our approach is not relevant for such executions. Applying our approach for program executions that exceed cache size is discussed in future work.

Average TCD is a good indicator of performance improvements that can be achieved with `Approx` ordering when test executions fit in the cache. A higher average TCD indicates the differences in instructions executed by test cases in the test suite is higher.

Ordering for instruction cache locality has a higher impact on performance gains for such test suites since it ensures that tests with high TCD between them are not executed in succession, avoiding cache misses that result from the difference. LLVM Symbolizer has the highest average TCD among subject programs of 42.18% and also the highest performance improvement with `Approx` of 13.06%. Test suites with low average TCD contain tests that execute largely the same set of instructions (similar control flow). This is often seen in programs, such as those in the EEMBC suite, that are largely sequential in their control flow with only a few control flow statements. As a result, any order will have high instruction locality. `Opt` and `Approx` orderings will not result in any significant improvements for such programs and test suites. For our subject programs, we found that when average TCD was low ($< 2\%$), execution time improvement was correspondingly low ($< 2\%$).

5.5.1 Recommendations

Based on the results over our subject programs, we recommend the `Approx` ordering of test cases in a test suite since it achieves (1) Comparable execution time improvements to `Opt`, and (2) Scales well to large numbers of tests, as opposed to `Opt`. Overhead of `Approx` is less than that of `Opt` for large test suites and can be further reduced by running the algorithm on GPUs. For subject programs whose executions fit in the cache, we found average TCD serves as a good guide for determining whether `Approx` ordering will result in reasonable performance improvements.

5.5.2 Effect on Fault Finding Capability

While our proposed test orderings reduce the overall test suite execution time, they have a negative effect on the *progressive* fault finding capability of the test suite (i.e. the ability of the test suite to uncover a bug quickly during execution). The *test coverage* in our proposed orderings increases very slowly since every subsequent test case executes as similar code (of the software under test) as possible with its previous one. Therefore, a bug that is uncovered by a test case that differs significantly (according to the 5.2 equation) from the first test case (in our ordering), would be identified at the later stages of the test suite execution. The exact opposite is true for a `BC` test ordering since every subsequent test case would be executing as *different* code as possible compared to its previous one. However, as seen in section 5.3, `BC` test orderings tend to be significantly slower.

The *overall* fault-finding capability is not affected by our proposed orderings since the test cases themselves are not altered. Once the test suite has finished, the fault finding capability would be the same as a `BC` or any other test case ordering. Therefore we recommend using our orderings in cases where the completion of the test suite execution is feasible in order to benefit from time reduction without sacrificing fault finding capability. In cases where its infeasible to execute all the test cases of a test suite every time, we recommend the following approach:

1. Optimise for test coverage (e.g. `BC`).
2. Identify the first X test cases from the ordering of step 1 which are feasible to execute every time.
3. Apply our proposed orderings to the identified set of X test cases.

5.5.3 Future Work

The effectiveness of proposed orderings, as with compiler optimisation techniques, depends on the characteristics of the program and test cases. Size of the program, distances between test case runs, number of test cases, cache size, will all have a significant effect on the performance gained from our approach. For programs whose executions exceed cache size, ordering of tests will have little effect since mutliple executions do not fit in the cache. We discuss this challenge and a potential solution that we plan to pursue in our future work. We also discuss our approach in a parallel test execution setting.

5.5.3.1 Scaling with Size of Program

As program execution size increases, instruction locality across test runs becomes a challenge since the cache may not accommodate all the instructions from a single program run resulting in capacity misses. To tackle this challenge, we plan to explore splitting programs into segments that fit in the cache. For instance, let's say we split a program P into four segments, $S1$, $S2$, $S3$, and $S4$, such that each segment fits in the cache. We run all test cases on segment $S1$ storing the results, and then we run all test cases on segment $S2$ using the results from $S1$ and so forth. Storing and reading intermediate results from the segment run of a test case in order to execute the successor segment can be overlapped with the execution of other test cases on that segment, reducing the potential bottleneck it may cause. This is a classic pipelining

problem which has a well known solution with respect to instructions. We plan to suitably adapt existing ideas for instructions to segments. Running permuted test cases on the segments rather than the whole program may help leverage instruction locality for large programs. We will explore the merits of this approach in our future work.

5.5.3.2 Running Tests in Parallel

It is often the case that test suites with large numbers of test cases are not run sequentially on a single processor, and are, instead, launched simultaneously on multiple processors. To achieve this, the test suite is split into groups (or collections) and each group of tests is executed on a different processor. Our algorithm for permuting test cases works by creating groups of tests with low distances within them. Every time we pick a new starting test case (Step 4 in `Opt` and `Approx` algorithms), we start a new group. We could, therefore, easily use our approach to create and launch groups of tests, with potentially higher instruction locality, on multiple processors. We believe our approach holds promise of time savings for executions on multiple processors; we will evaluate this hypothesis in our future work.

5.6 Summary

We presented an approach for ordering test cases to increase cache locality across test executions. We conducted empirical evaluations to assess execution time savings using the original approach and approximation when compared to random orderings and an ordering maximising branch coverage for programs from SIR, EEMBC benchmarks and an LLVM Symboliser.

Our evaluations revealed that ordering test executions to maximise instruction locality **improves** execution time. The nature of programs and test cases, in terms of range of distances between test case executions, determine the magnitude of the effect. The differences between worst and best random permutations ranged between 8.53% to 29.48% over SIR programs and 27.12% for LLVM Symbolizer providing evidence that order *matters* for test executions. Among the different orderings, `Opt` was best performing but could not scale beyond 14K tests for EEMBC programs. `Approx` was able to scale to large numbers of tests and perform comparably to `Opt`. Performance improvement with `Approx` over `BC` was a maximum of 9.3% for SIR programs, 2.94% over EEMBC and 13.06% over LLVM Symbolizer. Based on our results, it is clear

that reducing cache misses with `Approx` ordering can result in substantial performance gains. Average test case distance can be used as a guide for determining whether `Approx` ordering will result in reasonable performance improvements.

Chapter 6

Device-Based Test Case Scheduling for Heterogeneous Test Suites

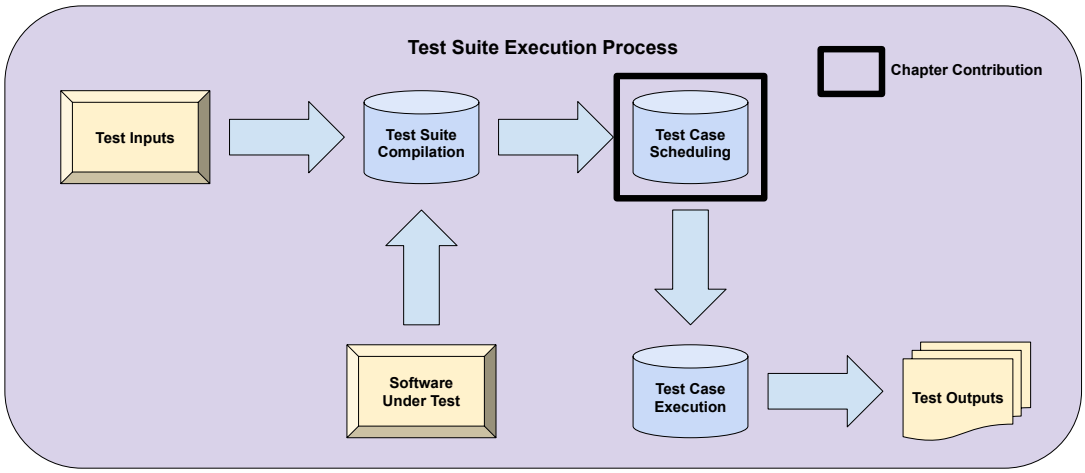

Figure 6.1: Chapter Contribution

In this chapter we target again the test case scheduling step (see figure 6.1) and explore, for the first time, device-based test scheduling for reducing the execution time of heterogeneous test suites. We propose and assess a minimal-overhead test scheduling algorithm which schedules the test case executions depending on their target device. The goal is to identify whether load balancing between the devices of a heterogeneous system can reduce the **overall** execution time of a heterogeneous test suite. We perform our assessment on an large-scaled test suite being developed by Codeplay, in tandem with ComputeCPP™, which validates implementations of the SYCL standard. We found that our scheduling algorithm can offer a maximum of 25.42% improvement in the execution time of a heterogeneous test suite when compared to parallel test

scheduling which does not consider the target device of test cases.

In the previous chapter we presented a series of test scheduling algorithms that reduce the overall execution time of test suites by improving instruction cache locality. That approach is applicable on single-core and multi-core architectures. It is not applicable, however, on heterogeneous architectures due to the fact that heterogeneous applications are massively data driven (e.g. graphics rendering) with some widely used devices like GPUs not having a cache memory at all. Moreover, as stated in section 2.5.2, the problem of executing N test cases is expanded in the heterogeneous domain into executing $N * M$ cases with M being the number of devices we wish to verify our application against. In heterogeneous test suites every test case execution is associated with a target device.

SYCL [57] is an open standard which is based on top of OpenCL [185] and defines a high-level C++ programming model for programming heterogeneous architectures. Codeplay has developed an implementation of the SYCL standard, ComputeCPP™, which comprises of both a runtime library and a compiler. ComputeCPP™ covers a wide range of features and hardware capabilities for effectively utilizing system resources. Testing such a complex system, however, requires a very large number of test cases for achieving high levels of test coverage. This, in turn, has led to lengthy test suite executions for ComputeCPP™ which have a negative impact on its life-cycle.

The rest of the chapter is organized as follows: Section 6.1 provides a background on the SYCL standard as well as the test suite used for the assessment. In section 6.2 we illustrate our initial device-agnostic test scheduling methodologies that form the baseline of this experiment. Section 6.3 presents our device-based test scheduling algorithm. Our experimental methodology and results are described in section 6.4. Finally, section 6.5 discusses threads to validity, limitations as well as future work and section 6.6 concludes.

6.1 SYCL

SYCL is a royalty-free open standard which defines a high-level single source C++ programming for programming heterogeneous architectures such as GPUs, FPGAs, DSPs and other kinds of accelerators, that is produced by the Khronos Group [56]. The latest version of this standard is SYCL 1.2.1 and was released in December 2017.

SYCL is based on top of OpenCL, another standard from the Khronos Group which defines a low-level C API and C based kernel language for writing heterogeneous

applications. SYCL provides the same performance portability and access to OpenCL hardware as with traditional OpenCL however, it also provides a high-level interface which removes much of the boilerplate code typical to OpenCL, but also provides many additional features such as data dependency analysis and task scheduling.

SYCL is single source, which means that rather than having the host side code and the kernel code (code which is compiled for the heterogeneous architecture) defined separately as is the case with OpenCL, both the host code and kernel code exist in the same C++ source file. This opens up users to a range of benefits such as stronger type safety between host and device and the ability to create templated kernel code. The ability to create template kernel code allows users to create generic library code as well as to create compile-time DSELs where an expression can be composed together into a single type. This technique can be used to fuse multiple kernel functions together removing the overhead of invoking kernel functions and moving data that can dramatically improve the performance of applications.

Finally, SYCL provides some high-level mechanisms which make programming applications for heterogeneous architectures much more accessible. Data dependency analysis takes the data requirements provided in an application and implicitly detects which tasks are dependent on other tasks. A runtime scheduler then enqueues tasks efficiently depending on the data dependency analysis. Finally, a fallback mechanism is provided which can allow an application to recover from failures and continue executing.

6.1.1 HammerSYCL

To compliment their implementation of the SYCL standard, Codeplay are developing HammerSYCL, a robust and extensive test suite whose aim is to test implementations of the SYCL standard far beyond from what is covered by the Conformance Test Suite provided by the Khronos group, which is limited in what it can cover. HammerSYCL tests not only the validity of the SYCL interface but also the C++11 and kernel language features, passing of kernel arguments, asynchronous execution, error handling and interoperability with other frameworks. It also provides both negative and positive testing, tests for extended functionality and extensive combination testing of the various features of SYCL in order to stress test the system. The test suite developed is used to evaluate ComputeCPP™ as it currently is, and all future versions as they develop.

HammerSYCL tests SYCL implementations in two ways: Firstly, with a large col-

lection of tests that cover all elements of the SYCL specification and secondly, by generating pseudo-random tests based on a heuristic model of the SYCL runtime; specifically the memory and execution model, that stress test various combinations of SYCL features which can be used together to produce a desired result. The number of test cases HammerSYCL is able to generate increases rapidly as its automatic test case generation functionality gets enhanced. This has a positive impact on its fault-finding capability but it results in test suites with long execution time.

6.2 Device-Agnostic Test Case Scheduling

At the early stages of our test suite development, performance was not an issue since the number of test cases was relatively low. We, therefore, opted for sequential test case scheduling (see figure 6.2) where every test case would be executed against all devices in a sequential fashion before the framework would move on to the next test case.

The sequential approach of figure 6.2 soon became a bottleneck to our continuous integration pipeline as the number and complexity of tests increased. We, therefore, opted for a coarse *device-agnostic* model for test parallelization where a *single test execution* was equivalent to the execution of a test against **all** target devices. In other words, when a test was chosen to execute inside a thread provided by the testing framework, it meant that the test would execute against **all** target devices sequentially inside that thread. Parallelization was achieved by having multiple threads executing different tests in parallel. Our coarse *device-agnostic* test scheduling is illustrated in figure 6.3: In a testing framework with 8 threads available, we would have 8 tests (one per thread) executing in parallel and inside each thread, the corresponding test would be executing against **all** devices sequentially.

6.3 Device-Based Test Case Scheduling

Having to execute a test against multiple targets (devices) for ensuring that a heterogeneous application is able to support them leads to a big increase in the execution time of a heterogeneous test suite every time a new test is being added. In an environment with M devices, every new test developed results in M new test executions. The *device-agnostic* test scheduling presented in section 6.2 was sufficient until we introduced automatic test generators in our testing framework, a feature that increased

	Device 1 (D1)	Device 2 (D2)	...	Device M (DM)
Test 1 (T1)	T1 on D1	T1 on D2	...	T1 on DM
Test 2 (T2)	T2 on D1	T2 on D2	...	T2 on DM
...	...	...	...	...
Test N (TN)	TN on D1	TN on D2	...	TN on DM

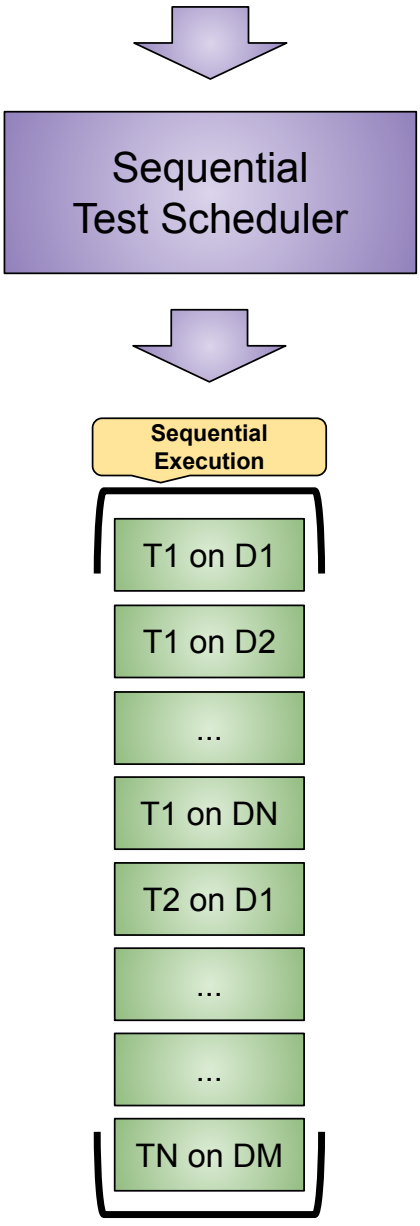

Figure 6.2: Sequential Test Case Scheduling

	Device 1 (D1)	Device 2 (D2)	...	Device M (DM)
Test 1 (T1)	T1 on D1	T1 on D2	...	T1 on DM
Test 2 (T2)	T2 on D1	T2 on D2	...	T2 on DM
...	...	...	...	...
Test N (TN)	TN on D1	TN on D2	...	TN on DM

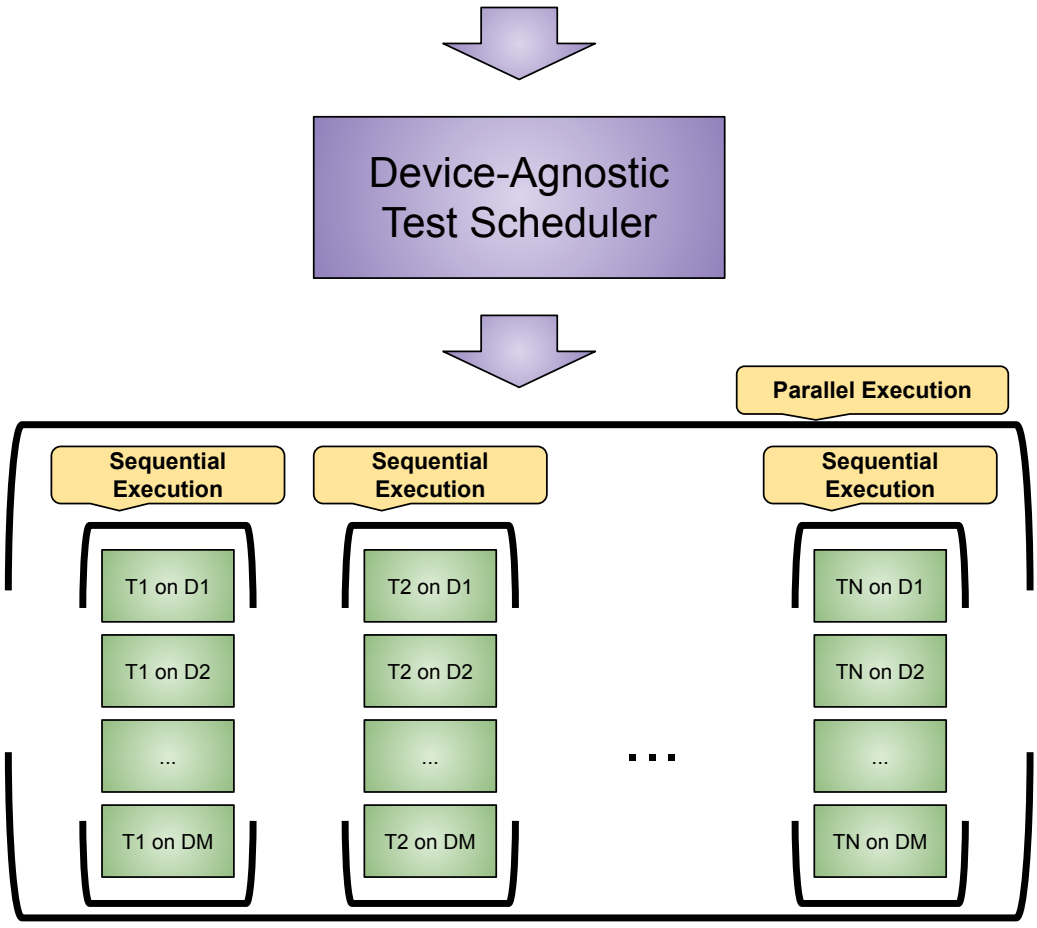

Figure 6.3: Device-Agnostic Test Case Scheduling

the number of tests in the test suite exponentially. This rapid increase of tests started having a negative impact to our product life-cycle.

To make the use of our test suite inside our continuous integration pipeline feasible again, we extended our testing framework and implemented a more *granular* test scheduler which takes into consideration the *target device* of each test. Given a set of *(test, target)* pairs, the new *device-based* test scheduler is responsible for scheduling their execution. This extension allowed us to explore the degree to which test scheduling based on the *target device* is able to reduce the **overall** execution time of a heterogeneous test suite.

The *device-based* test scheduler uses a minimal-overhead algorithm which ensures that each device has a **specific** number of tests executing in parallel against it **throughout** the test suite execution. This number can be different for each device and depends on the throughput capability of the device, the limitations of the device driver as well as the maximum numbers of tests that the testing framework allows to execute in parallel.

Our approach is illustrated in figure 6.4: Given N tests that need to be executed against M devices, our test scheduler ensures that, at any given time during the test suite execution, there will be (for example) 3 tests executing against device *D1*, 2 tests executing against device *D2*, 2 tests executing against device *DM* etc. The number of tests associated with each device (3, 2 and 2 in our example) is passed as input to the test scheduling algorithm and their sum must not exceed the maximum number of tests that the testing framework is allowed to execute in parallel.

The core idea behind this *device-based* test scheduling is that, at any point during the execution of a heterogeneous test suite, each device will be accepting computational payloads from a **specific** number of tests. This parallel test execution model reduces significantly the imbalance between the processing units of a heterogeneous system during the execution of a test suite, something which has the potential to reduce its **overall** execution time. For example, in the scenario where, at a certain point during test suite execution, the vast majority of the executing tests target the CPU of the system, the GPU will be nearly idle while the CPU overloaded with computational tasks. In a situation like this, having less tests executing against the CPU would allow it to process the remaining tests faster and, at the same time, having more tests executing against the GPU would offer additional speed-up in the overall test suite execution.

	Device 1 (D1)	Device 2 (D2)	...	Device M (DM)
Test 1 (T1)	T1 on D1	T1 on D2	...	T1 on DM
Test 2 (T2)	T2 on D1	T2 on D2	...	T2 on DM
...	...	...	...	...
Test N (TN)	TN on D1	TN on D2	...	TN on DM

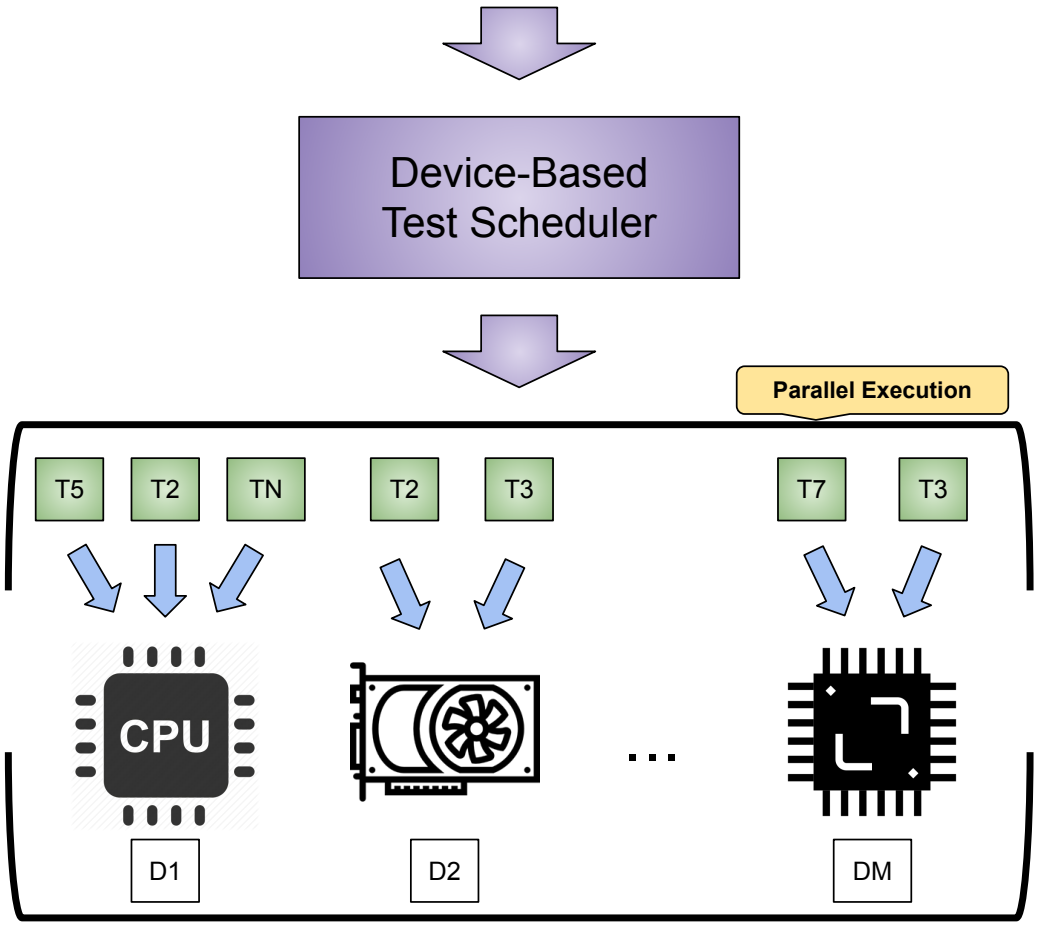

Figure 6.4: Device-Based Test Scheduling

6.3.1 Test Case Scheduling Algorithm

Algorithm 4 illustrates our device-based test scheduling: Given a heterogeneous system with M devices, N tests to execute, PT tests that the testing framework will be executing in parallel as well as the number of tests associated with each device, our algorithm first ensures that the sum of tests to be run in parallel for each device is less or equal to the total number of tests that the framework will be executing in parallel (line 1). If this condition is false (i.e. $\sum_{i=1}^{M} DTi > PT$), it means that the testing framework does not provide enough threads. Then, for each device i, we create a set containing all N tests (TCS_i on line 4) and we initiate DTi test executor threads which start to execute (in parallel) the tests in TCS_i against device i. As shown in algorithm 5, each test executor will keep executing tests against device i (and removing them from TCS_i) until there are no more tests in TCS_i.

In a scenario where we have $N = 10$ tests to be executed against $M = 2$ devices, a testing framework that will be executing $PT = 5$ tests in parallel (in total - independently of the target device), 3 tests associated with device 1 (DT_1) and 2 tests associated with device 2 (DT_2), algorithm 4 will first ensure that $DT_1 + DT_2 \leq PT$ (i.e. $3 + 2 \leq 5$). Then, for device 1, a test case set will be created ($TCS_1 = \{1,2,3,4,5,6,7,8,9,10\}$) and $DT_1 = 3$ test executor threads will be launched each of which will be executing tests in parallel from TCS_1 until TCS_1 is empty. At this point, its important to emphasize that all 3 test executor threads will be accessing the **same** set TCS_1 (i.e. they won't be operating on copies). Finally, the same procedure will be repeated for device 2 but this time only $DT_2 = 2$ test executor threads will be launched.

6.4 Experiment and Results

We execute HammerSYCL and measure its **overall** execution time for the following test schedulers:

- **Sequential Test Scheduler** - see figure 6.2.
- **Device-Agnostic Test Scheduler** - see figure 6.3.
- **Device-Based Test Scheduler** - see figure 6.4.

We repeat our experiment for a different number of threads that the testing framework will be executing in parallel (PT input in algorithm 4). This number defines how many tests will be executing in parallel independently of their target device. Due to technical

Input: N test cases, M devices, PT the number of tests that the testing framework will be executing in parallel, $DT_i \forall i \in \{1, M\}$ the number of tests associated with each device.

Output: Test suite completed.

1: If $\sum_{i=1}^{M} DTi \leq PT$, go to step 3.
2: Insufficient framework threads. Inform the user and **EXIT**.
3: **for all** $i \in \{1...M\}$ **do**
4: $TCS_i = \{1...N\}$. // N test set for device i.
5: **for all** $j \in DT_i$ **do**
6: TET_j = create a test executor **thread** for device i.
7: TET_j.run(i, TCS_i). // See algorithm 5 - TCS_i passed by **reference**.
8: **end for**
9: **end for**
10: Await all PT test executor threads to finish.
11: **EXIT**.

Algorithm 4: Device-Based Test Scheduling Algorithm

Input: i the device to execute the tests against, RTC (**synchronized** access) remaining test cases.

Output: Test cases for device i completed.

1: **while** RTC is not empty **do**
2: j = **pop** the first element from RTC.
3: Execute test case j against device i.
4: **end while**
5: **EXIT**.

Algorithm 5: Test Executor Algorithm

restrictions of the device drivers, the upper limit of threads that our testing framework is allowed to have is 64.

6.4.1 Measurement

We run our experiments using a desktop computer powered by an Intel Quad Core 6700 processor at 3.4 GHZ with 128KB of Instruction Cache and 128KB of L1 data cache. The system also included an AMD Radeon GPU 5450 series with 80 stream processors. The machine runs Ubuntu 14.04 with Linux kernel 3.16.0.33. We measure the execution time of the whole test suite execution using python's *time.time()* command. We repeated each measurement 10 times and report the average value. We do not report the standard deviation as it was less than 0.4% for **every** measurement.

6.4.2 Results

Figure 6.5 contains the execution time of the HammerSYCL test suite with *device-agnostic* and *device-based* parallel test scheduling for an increasing number of threads available to the testing framework. Figure 6.6 illustrates the speed-up achieved when comparing *device-agnostic* and *device-based* parallel test scheduling against sequential test scheduling (see figure 6.2). It is clear that *device-based* test scheduling can significantly reduce the execution time of heterogeneous test suites. By applying the *device-agnostic* test scheduling we were able to achieve an execution speed-up in the range of 17% to 36% when compared to sequential test scheduling with the highest speedup being achieved when having 64 threads available in the testing framework. After implementing our *device-based* test scheduler we were in a position to achieve speed-up ranging from 30% to 47% with the 47% being achieved with the framework having 64 threads at its disposal. For the 64 thread case, we experience an average of **20.3% improvement** in the execution time of HammerSYCL when we apply our *device-based* scheduling algorithm. Finally, it is worth noting that the *device-based* scheduling outperforms the *device-agnostic* scheduling across **all** our experiments.
For obtaining the results presented in figures 6.5 and 6.6 we assigned the **same** number of tests to each one of our available devices (an Intel CPU and an AMD GPU). This means that in the case where we achieved maximum speed-up (64 threads available to the testing framework) there were 32 tests executing in parallel against the CPU and 32 tests executing in parallel against the GPU at **any** point in time during the execution of the test suite. This brings the question of whether we could have achieved even

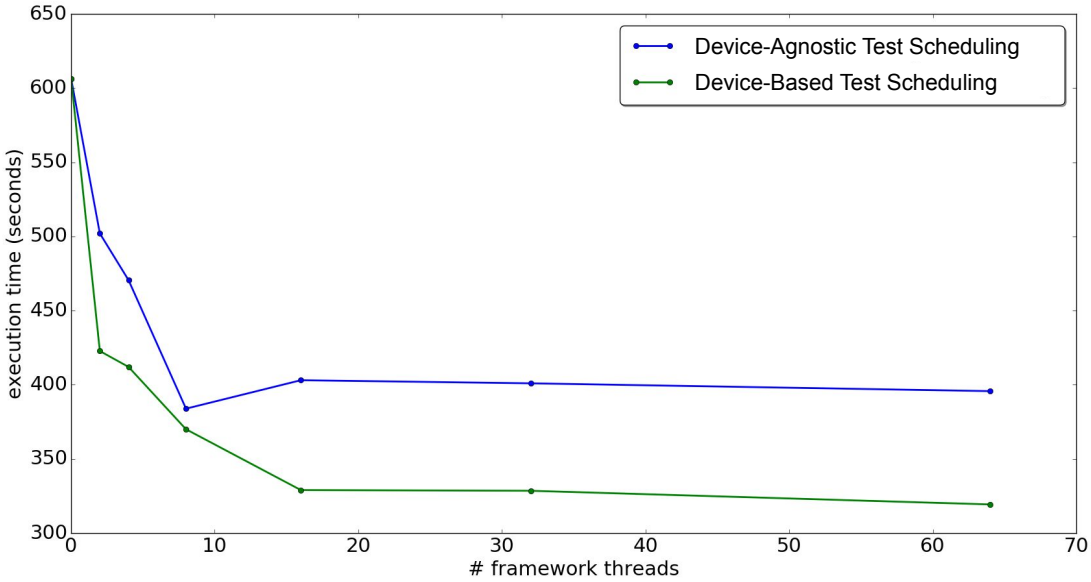

Figure 6.5: HammerSYCL Execution Time with Device-Agnostic and Device-Based Parallel Test Scheduling

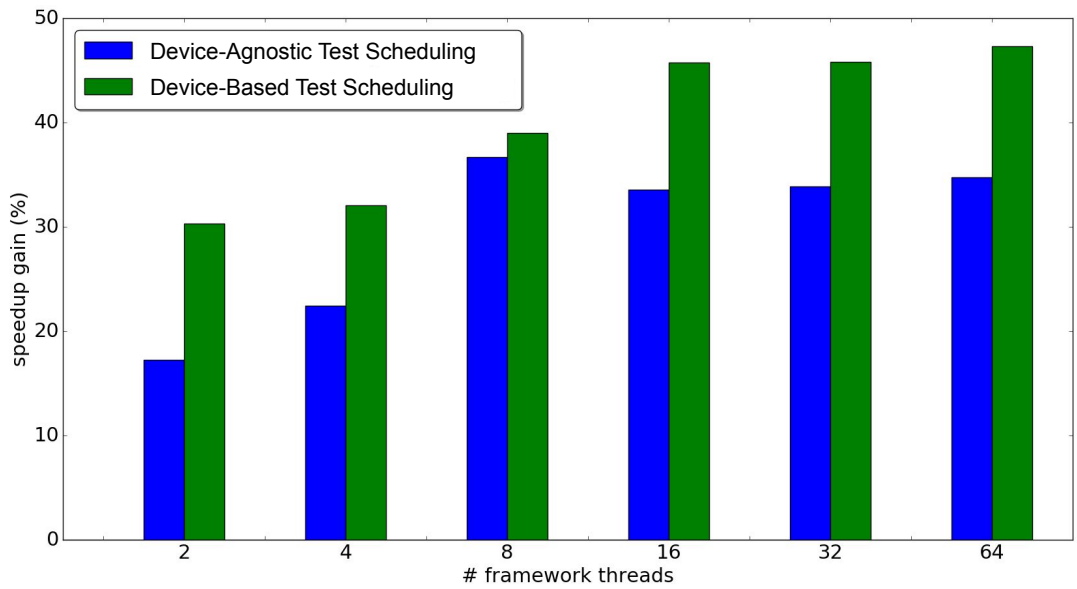

Figure 6.6: HammerSYCL Execution Speed-Up of Parallel Test Schedulings over Sequential Scheduling

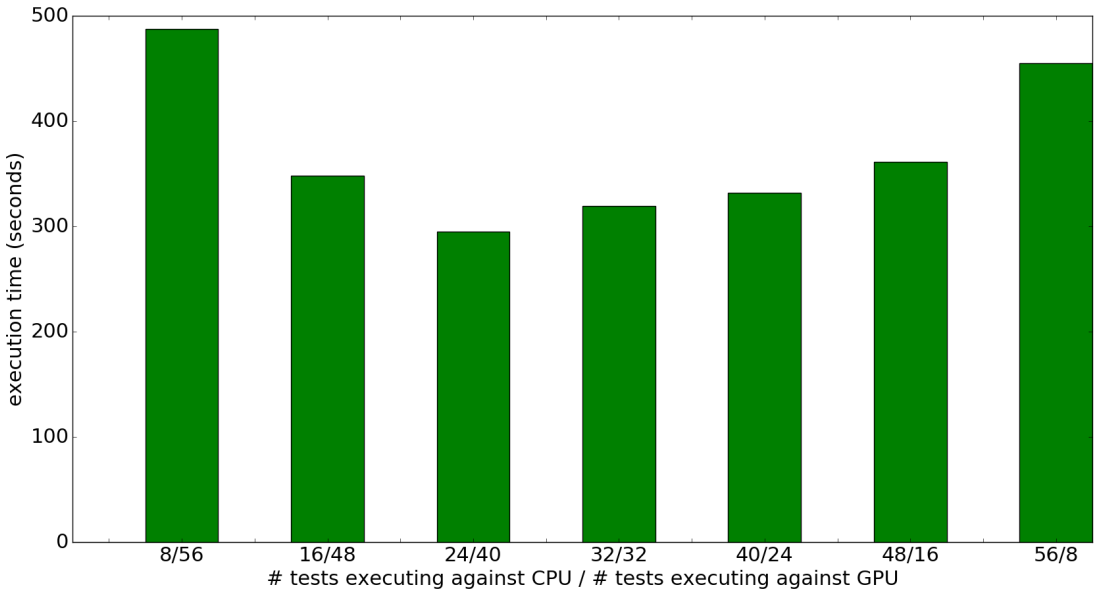

Figure 6.7: HammerSYCL Execution Time for Various Test Distributions Across Devices

better speed-up by distributing the 64 framework threads in a different way between the devices. Figure 6.7 contains the execution time of HammerSYCL when executed in a framework with 64 available threads but with various distributions of these threads between the devices. Having 24 tests executing in parallel against the CPU and 40 tests executing in parallel against the GPU **throughout** the execution of the test suite achieves the best speed-up which is translated to a **25.42% improvement** when compared to parallel test execution without scheduling. This can be explained from the fact that, in our experimental setting, the CPU of the system executes **also** the host code of the tests therefore having less tests executing their device code against it in parallel actually improves its throughput.

6.5 Discussion

6.5.1 Overhead

The overhead of our test scheduling is minimal because the scheduling algorithm does **not** depend on the number of the test cases in the test suite. Our algorithm does depend, however, on the number of devices of the heterogeneous system under test for initiating the executor threads assigned to each device. The asymptotic complexity of our proposed test scheduling is $O(M)$ with M being the number of system devices.

6.5.2 Algorithm Parameters

Our algorithm accepts as parameters the N test cases to execute, the M devices to execute the tests against, the number of threads the testing framework will be executing in parallel (PT) as well as the number of tests that should be executing in parallel against each device (DT_i for device i). Defining the first three parameters of the algorithm is straightforward: N represents the tests we want to execute, M represents the target devices we want to execute the tests against while PT reflects the number of tests we want to execute in parallel. On the other hand, associating a test case number to each device is non-trivial since the factors that influence how many tests each device should be executing in parallel are related to the device capabilities as well as its driver limitations. As illustrated in figure 6.7, the DT_i algorithm parameters have a big impact on the execution time of a heterogeneous test suite. Therefore, we recommend an experimentation process which involves executing the test suite on the same system but with different DT_i parameters in order to identify the set of parameters that achieve the highest speedup. This experimentation process needs to be done once for a given system: after we obtain the ideal set of DT_i parameters we can re-use them in subsequent test suite executions. For systems with many devices, we recommend assigning each device the **same** number of tests as an approximation.

6.5.3 Threats to Validity

We identify one threat to external validity of our assessment based on the test suite selection. We use for our experiment a developer created test suite for an industrial project. Despite the fact that it is an extensive test suite for testing a combination of a compiler and a runtime, we cannot claim that it is necessarily representative of all possible heterogeneous test suites. In fact, the additional speed-up of our test prioritization depends **highly** on the proportion and complexity of device code as opposed to host code of the tests. The overhead of host code execution is unavoidable when executing multiple heterogeneous tests in parallel. Therefore, test suites that have computationally expensive host code and trivial device code will show minimal additional speed-up by using our proposed execution model. The opposite is true for test suites with minimal host code and computationally expensive device code.

6.5.4 Limitations

We also identify a limitation that is related to the framework we use for executing the test cases. By being able to specify the target device for a test we achieve, on the one hand, high levels of rigorousness (because the device code of **every** test is executed on **every** device we wish to test) but, on the other hand, we do not test certain features of the SYCL runtime. More specifically, the SYCL runtime includes a scheduler which enqueues computational tasks efficiently depending on data dependency analysis and is responsible not only for the order of the task execution but also for the choice of device that each task should execute on. By specifying the target device for each test we do not provide the SYCL runtime with the freedom of device choice and, consequently, its task scheduling functionality is not verified. It is, therefore, essential for the testing framework to provide an **additional** test execution mode where multiple devices would be available to the SYCL runtime during test execution. However, when the SYCL runtime has the ability to chose where to execute the device code of a test, our device-based test scheduling approach is not applicable.

6.5.5 Future Work

As described in section 6.3, our current test scheduling algorithm is based **entirely** on the target device of each test. However, in our current test execution setting, we have a collection of *(N * M) (test, device)* pairs (N being the number of tests, M the number of target devices) to execute as independent processes. This brings the question of whether our algorithm can improve by taking into consideration additional information about the tests during scheduling. This information could be static and/or dynamic and can be obtained by previous executions of the test suite. Our early thoughts regarding this include the characterization of each *(test, device)* pair depending on how computational intensive it is (i.e. an indication of how much a test stresses a specific device). This can be obtained by recording the execution time of the test against that device during a previous test suite run. Given this complexity information, our scheduling algorithm could potentially ensure that a device would never execute multiple computational intensive tests in parallel, thus avoiding contention. For example, by using a hypothetical scale from 0 to 10 for characterizing the complexity of a test executed on a device (10 being the most computational intensive), our algorithm could attempt to schedule the tests in such a way that it is never the case (during the test suite execution) for a device to be executing tests of **combined** complexity greater than 15 in parallel.

At the same time though, this approach will have a negative impact on the overhead of test scheduling since the algorithm will depend **additionally** on the number of test cases in the test suite.

6.6 Summary

We have presented a novel approach for scheduling heterogeneous system tests depending on their target device. Our approach increases the load balance between the devices of a heterogeneous system during test suite execution leading to improved execution time. We evaluated our test scheduling algorithm on HammerSYCL, an extensive test suite developed by Codeplay Software that tests implementations of the SYCL standard. Our device-based test scheduling methodology achieved an average of **25.42%** speed-up when compared to the device-agnostic approach we were using before for executing the HammerSYCL tests. Finally, the overhead of our test scheduling algorithm is insignificant.

As we move from the multi-core to heterogeneous computing, there is a growing need for adapting our current testing frameworks and methodologies. Our future work in Codeplay includes not only the enhancement of our current test scheduling algorithm but the implementation of a unified testing framework for heterogeneous applications that make use of open standards like OpenCL and SYCL. We plan to tackle the challenges of heterogeneous test generation, compilation, execution and reporting and include our approaches in a modular heterogeneous testing framework.

Chapter 7

Conclusion

In this thesis, we proposed a series of techniques that achieve significant time reduction for test *workflows* without the need to remove any test cases or to upgrade computing infrastructure. Our motivation was the fact that modern software systems are so large and complex that the number of non-redundant test cases needed for effective validation is extremely large and has a negative impact on development productivity. Our proposed techniques can be divided into a data transformation which improves test suite compilation time and a series of test case scheduling algorithms that produce time efficient test case execution orders.

Test suites are being treated by industry and academia as abstract *artifacts* in which test cases are added or removed in order for a balance between test feasibility and effectiveness to be achieved. This thesis followed a different approach and explored, for the first time, ways of reducing test *workflow* time without any loss of information (i.e. without removing or redacting test cases) or an infrastructure change. We started by acknowledging the fact that the test suite execution process entails three major phases (*test suite compilation*, *test case scheduling* and *test case execution*) and continued by presenting the challenges for these steps as the test case number increases. We then proposed three techniques which result in significant time reduction of the *test suite compilation* and *test case execution* phases.

For the *test suite compilation* phase, we proposed a test code transformation, as a pre-compilation step, that significantly reduces the compilation time and also enables the inclusion of more test cases in a test suite. Our approach restructures the *test inputs* as well as the calls to the *software under test* and provides a semantically equivalent test suite which can be compiled up to 69 times faster and is able to include 10 times more test cases. We evaluated the transformation against two widely used compilers (GCC

and Clang) and a variety of benchmarks including an industrial application developed by Codeplay Software.

We then targeted the *test case scheduling* phase and proposed a series of test case scheduling algorithms for the single-CPU and multi-CPU architectures which produce execution orders that maximise instruction cache locality across test case executions. The scheduling algorithms perform nearest neighbour analysis on the test cases, which are represented by their executed instructions, and produce execution orders where test cases that execute similar instructions are executed consecutively. Increased instruction cache locality leads to fewer instruction cache misses which, in turn, leads to a faster *test case execution* phase. We evaluated our algorithms on 20 benchmarks including an LLVM tool. Our results suggest that the effect on the *test case execution* time depends highly on the nature of the *software under test* as well as the test cases with speed-ups ranging from 0.48% to 29.48%.

In our final contribution we targeted again the *test case scheduling* phase and proposed, for the first time, a test case scheduling algorithm for heterogeneous architectures. Device load balancing during the *test case execution* phase was achieved by scheduling test cases depending on their target device. Ensuring that each device will be accepting computational payloads from a specific number of tests during the execution of a test suite minimizes device idle time as well as device overloading. This work was conducted in collaboration with Codeplay Software for reducing the *test case execution* phase of a large-scaled, industrial test suite targeting Codeplay's in-house implementation of the SYCL standard. The maximum speed-up achieved is 25.42% (average 20.3%) while the test scheduling overhead was insignificant.

7.1 Putting Everything Together

Our contributions optimize two phases of the test suite execution process as defined in chapter 1: contribution **C1** optimizes the *test suite compilation* phase and contributions **C2** and **C3** optimize the *test case execution* phase (both by operating on the previous phase - the *test case scheduling*). We, therefore, identify two potential approaches that can be explored in future work which are based on the idea of **combining** the optimisations presented in this thesis for improving *both* the test suite compilation and test suite execution time:

- **Combination 1 -** Combine our data transformation (contribution **C1**) with instruction cache locality test case scheduling (contribution **C2**) for CPU-based

architectures. After the application of our data transformation proposed in chapter 4, every test case in our test suite is represented by a loop iteration. The order, however, of the test cases remains intact. We could then apply our instruction cache locality test case scheduling (see chapter 5) in order to optimize the execution time of the loop. Figure 7.1 presents the combined approach: The optimized test case scheduling is represented in lines 10 to 14 in the form of a one-to-one mapping of the loop iteration to the test case that needs to be executed next. In this particular example, the test case schedule that optimises instruction cache locality is: $\{1, 2, 0\}$. The test case loop in lines 20 to 24 remains largely intact except from the array indexing: instead of passing the iteration index i directly to the centralised data structures created as part of the data transformation (in order to execute test case i), we pass the map of the iteration index which defines the test case that needs to be executed next according to the test schedule optimizing instruction cache locality.

- **Combination 2 -** Combine our data transformation (contribution **C1**) with device-based test case scheduling (contribution **C3**) for heterogeneous architectures. As presented in chapter 4, our data transformation is applicable also to heterogeneous applications. The result of this transformation is identical to the one of CPU-based programs (i.e. a loop where each iteration represents a test case). We could then apply a simplified version of our device-based test scheduling (see chapter 6) in order to achieve device load balancing and thus optimize both test suite compilation and execution time. We cannot apply the full version of our device-based test scheduling because of the state our data transformation leaves the test code: all test cases are combined into a loop which means the test cases are executed inside a single *process* where we can specify only one target device. Figure 7.2 illustrates the simplified device-based test scheduling: The data transformation combines the N test cases into a combined test suite *CTS* which is then executed against every device in parallel. Under this model, each device would be executing a single test case at any given point in time during the execution of the test suite because *CTS* is effectively a sequential loop that executes one test case at a time. We could potentially instruct our data transformation to break the *CTS* in chunks (e.g. 2 chunks of $\frac{N}{2}$ test cases each). By this way we would be able to execute more than one test case in parallel against each device of the system. However, this approach would have a negative impact on

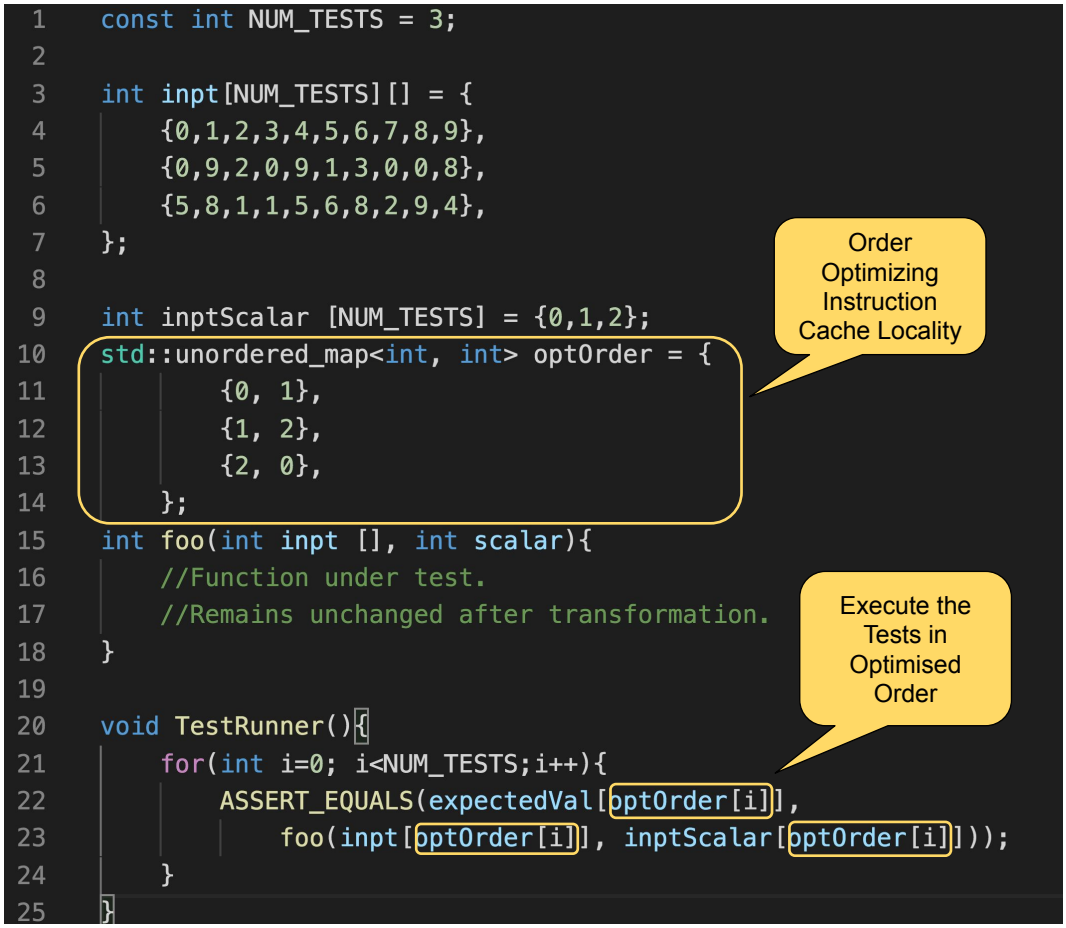

Figure 7.1: Data Transformation Combined with Instruction Cache Locality Test Case Scheduling

the compilation speedup since the resulting test code would be larger. Exploring this *trade-off* between our data transformation and device-based test scheduling (i.e. a *trade-off* between test suite compilation and execution speedup) should be part of the research on this combination.

7.2 Final Remarks

The research question we attempted to answer in this thesis, as defined in chapter 1, is the following:

> *Given a test suite, is it possible to reduce the time of the test suite execution process without any loss of information or a change on the underlying infrastructure?*

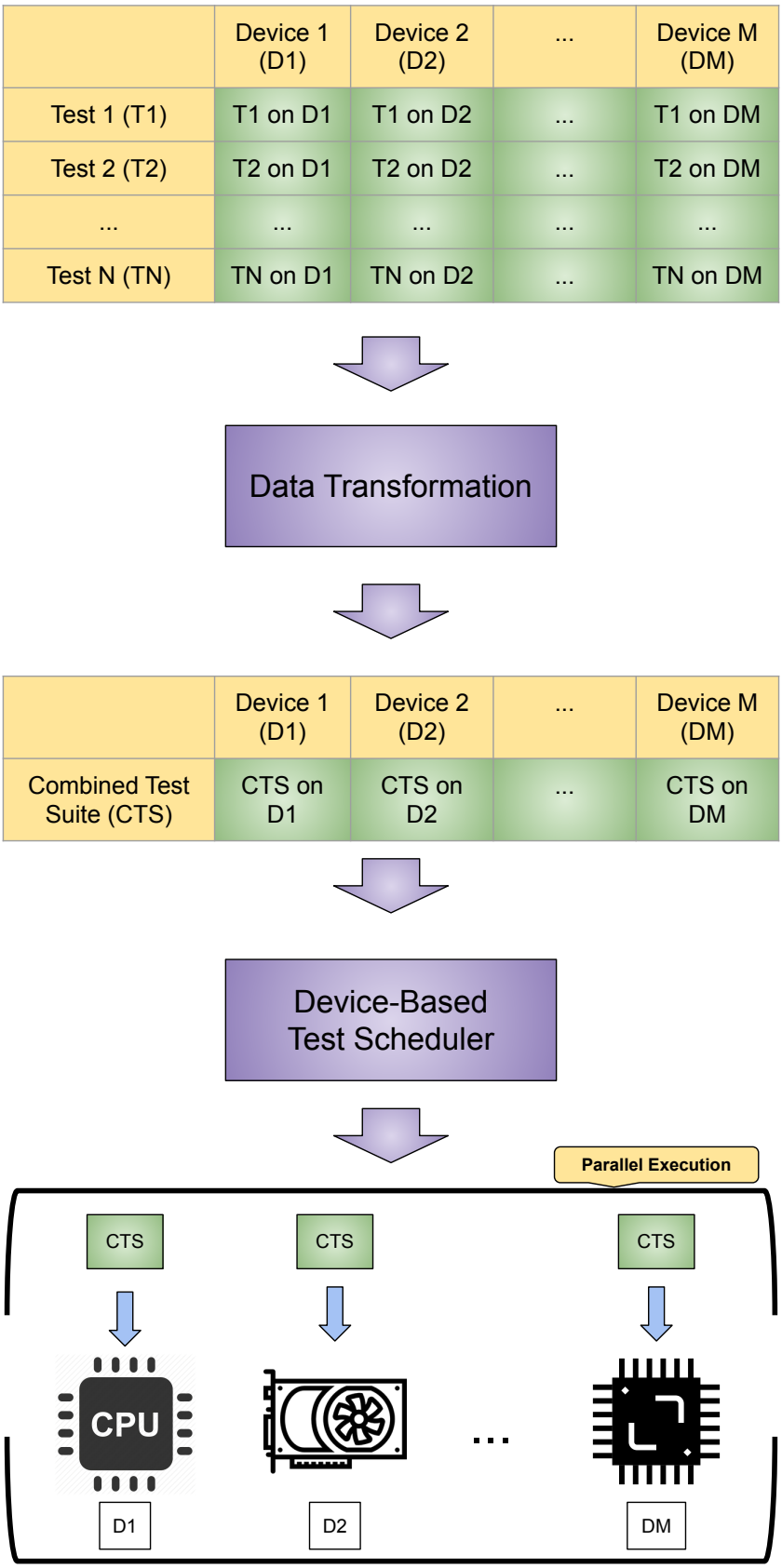

Figure 7.2: Data Transformation Combined with Device-Based Test Case Scheduling

Given the three contributions of this thesis, our answer to the above question is **yes** - it **is** possible to reduce the time of the test suite execution process without losing any information or having to change computing infrastructure. In all three contributions we were able to reduce the time of test *workflows* by utilizing *low-level interactions* of test suites. In our first contribution we optimised the process of transforming a test suite into optimised machine code while in the second contribution we utilised the instruction cache memory of CPU efficiently. Finally, in our third contribution we achieved load balance between the physical heterogeneous devices. Considering *how* we were able to optimise test *workflows* in this thesis, we conclude the following:

> *Unique patterns of test suite low-level interactions can also be exploited for test workflow speedup.*

With the growing complexity and responsibility of software, the number of test cases needed for effective validation becomes intractable. Even after the application of traditional test suite optimization techniques, real-world test suites are extremely large and their frequent execution becomes infeasible. The techniques proposed in this thesis can be effectively combined with existing test suite optimizations in order to further reduce software testing cost. We believe that the architectural patterns of test suites should be utilized even more by the software testing community in the quest to make testing feasible.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *1993 Conference on Software Maintenance*, pages 348–357. IEEE, 1993.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[4] F. E. Allen. Control flow analysis. In *ACM SIGPLAN Notices*, volume 5, pages 1–19. ACM, 1970.

[5] E. L. Alves, P. D. Machado, T. Massoni, and M. Kim. Prioritizing test cases for early detection of refactoring faults. *Software Testing, Verification and Reliability*, 26(5):402–426, 2016.

[6] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[7] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 312–321. IEEE, 2013.

[8] O. Astrachan. Bubble sort: an archaeological algorithmic analysis. In *ACM SIGCSE Bulletin*, volume 35, pages 1–5. ACM, 2003.

[9] M. Baluda, G. Denaro, and M. Pezze. Bidirectional symbolic analysis for effective branch testing. *IEEE Transactions on Software Engineering*, 42(5):403–426, 2016.

[10] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. *ACM SIGSOFT software engineering notes*, 29(4):108–118, 2004.

[11] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[12] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[13] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.

[14] D. W. Binkley, M. Harman, and K. Lakhotia. Flagremover: A testability transformation for transforming loop-assigned flags. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):1–33, 2011.

[15] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[16] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester. Test suite prioritization by cost-based combinatorial interaction coverage. *International Journal of System Assurance Engineering and Management*, 2(2):126–134, 2011.

[17] S. Carr, K. S. McKinley, and C.-W. Tseng. *Compiler optimizations for improving data locality*, volume 28. ACM, 1994.

[18] C. Catal. On the application of genetic algorithms for test case prioritization: a systematic literature review. In *Proceedings of the 2nd international workshop on Evidential assessment of software technologies*, pages 9–14, 2012.

[19] J. Cavazos and M. F. O'Boyle. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*, 41(10):229–240, 2006.

[20] K. Chakrabarty. Test scheduling for core-based systems. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 391–394. IEEE Press, 1999.

[21] K. Chakrabarty. Test scheduling for core-based systems using mixed-integer linear programming. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 19(10):1163–1174, 2000.

[22] K. Chakrabarty, E. Marinissen, et al. Test access mechanism optimization, test scheduling, and tester data volume reduction for system-on-chip. *IEEE Transactions on Computers*, 52(12):1619–1632, 2003.

[23] P. K. Chan, M. J. Boyd, S. Goren, K. Klenk, V. Kodavati, R. Kundu, M. Margolese, J. Sun, K. Suzuki, E. Thorne, et al. Reducing compilation time of zhong's FPGA-based SAT solver. In *Field-Programmable Custom Computing Machines, 1999. FCCM'99. Proceedings. Seventh Annual IEEE Symposium on*, pages 308–309. IEEE, 1999.

[24] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software: Practice and Experience*, 22(5):349–369, 1992.

[25] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2011.

[26] Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1056–1065. IEEE, 2007.

[27] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press, 2002.

[28] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: A system for selective regression testing. In *Proceedings of 16th International Conference on Software Engineering*, pages 211–220. IEEE, 1994.

[29] Z. Chen, Y. Duan, Z. Zhao, B. Xu, and J. Qian. Using program slicing to improve the efficiency and effectiveness of cluster test selection. *International Journal of Software Engineering and Knowledge Engineering*, 21(06):759–777, 2011.

[30] ComputeCpp. Computecpp – accelerate complex C++ applications on heterogeneous compute systems using open standards., 2017. "https://www.codeplay.com/products/computesuite/computecpp".

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[32] P. J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

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

[34] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396, 2015.

[35] K. M. Dixit. The SPEC benchmarks. *Parallel computing*, 17(10-11):1195–1209, 1991.

[36] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, 2010.

[37] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[38] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4), 2009.

[39] S. Eghbali and L. Tahvildari. Test case prioritization using lexicographical ordering. *IEEE Transactions on Software Engineering*, 42(12):1178–1195, 2016.

[40] W. S. A. El-Hamid, S. S. El-etriby, and M. M. Hadhoud. Regression test selection technique for multi-programming language. In *2010 The 7th International Conference on Informatics and Systems (INFOS)*, pages 1–5. IEEE, 2010.

[41] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 329–338, 2001.

[42] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[43] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[44] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2014.

[45] C. Fang, Z. Chen, and B. Xu. Comparing logic coverage criteria on test case prioritization. *Science China Information Sciences*, 55(12):2826–2840, 2012.

[46] M. Felderer and I. Schieferdecker. A taxonomy of risk-based testing, 2014.

[47] K. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6, 1981.

[48] K. F. Fischer and F. KF. A test case selection method for the validation of software maintenance modifications. 1977.

[49] G. Fursin, M. F. O'Boyle, and P. M. Knijnenburg. Evaluating iterative compilation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 362–376. Springer, 2002.

[50] D. Gao, X. Guo, and L. Zhao. Test case prioritization for regression testing based on ant colony optimization. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 275–279. IEEE, 2015.

[51] D. Garg and A. Datta. Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference-Volume 135*, pages 61–68, 2013.

[52] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):25, 2016.

[53] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, pages 317–324. ACM, 1997.

[54] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[55] B. Gough and R. M. Stallman. An introduction to gcc for the gnu compilers gcc and g++. *Network Theory Ltd*, pages 35–46, 2004.

[56] K. GROUP et al. Khronos: Open standards for media authoring and acceleration, 2009.

[57] K. O. W. Group et al. Sycl: C++ single-source heterogeneous programming for OpenCL, 2015.

[58] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1240–1247. IEEE, 1998.

[59] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Software Testing, Verification and Reliability*, 6(2):83–111, 1996.

[60] R. K. Gupta, S. K. Janumahanthi, M. Nagesh, V. R. Somisetty, P. Thota, and V. K. Vb. End to end testing automation and parallel test execution, May 12 2015. US Patent 9,032,373.

[61] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[62] F. Haftmann, D. Kossmann, and E. Lo. Parallel execution of test runs for database application systems. In *Proceedings of the 31st international conference on Very large data bases*, pages 589–600, 2005.

[63] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering*, 42(5):490–505, 2015.

[64] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–31, 2014.

[65] M. Harman. Refactoring as testability transformation. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 414–421. IEEE, 2011.

[66] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[67] M. J. Harrold. Testing: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000.

[68] M. J. Harrold and M. Souffa. An incremental approach to unit testing during maintenance. In *Proceedings. Conference on Software Maintenance, 1988.*, pages 362–367. IEEE, 1988.

[69] J. Hartmann and D. J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, 1990.

[70] M. P. Heimdahl and D. George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 176–185. IEEE Computer Society, 2004.

[71] H. Hemmati, A. Arcuri, and L. Briand. Reducing the cost of model-based testing through test case diversity. In *IFIP International Conference on Testing Software and Systems*, pages 63–78. Springer, 2010.

[72] H. Hemmati and L. Briand. An industrial investigation of similarity measures for model-based test case selection. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 141–150. IEEE, 2010.

[73] H. Hemmati, L. Briand, A. Arcuri, and S. Ali. An enhanced test case selection approach for model-based testing: an industrial case study. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, 2010.

[74] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th ICSE*, pages 483–493. IEEE Press, 2015.

[75] C. Hettiarachchi, H. Do, and B. Choi. Effective regression testing using requirements and risks. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 157–166, 2014.

[76] C. Hettiarachchi, H. Do, and B. Choi. Risk-based test case prioritization using a fuzzy expert system. *Information and Software Technology*, 69:1–15, 2016.

[77] R. M. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

[78] K. H. S. Hla, Y. Choi, and J. S. Park. Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 527–532. IEEE, 2008.

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

[80] S. Huang, Y. Chen, J. Zhu, Z. J. Li, and H. F. Tan. An optimized change-driven regression testing selection strategy for binary java applications. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 558–565, 2009.

[81] Y. Huang, W.-T. Cheng, C.-C. Tsai, N. Mukherjee, O. Samman, Y. Zaidan, and S. M. Reddy. Resource allocation and test scheduling for concurrent test of core-based soc design. In *Test Symposium, 2001. Proceedings. 10th Asian*, pages 265–270. IEEE, 2001.

[82] Y. Huang, S. M. Reddy, W.-T. Cheng, P. Reuter, N. Mukherjee, C.-C. Tsai, O. Samman, and Y. Zaidan. Optimal core wrapper width selection and SOC test scheduling based on 3-d bin packing algorithm. In *Test Conference, 2002. Proceedings. International*, pages 74–82. IEEE, 2002.

[83] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85(3):626–637, 2012.

[84] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[85] M. Z. Z. Iqbal, Z. I. Malik, M. Riebisch, et al. A model-based regression testing approach for evolving software systems with flexible tool support. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 41–49. IEEE, 2010.

[86] V. Iyengar, K. Chakrabarty, and E. J. Marinissen. Wrapper/tam co-optimization, constraint-driven test scheduling, and tester data volume reduction for socs. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 685–690. IEEE, 2002.

[87] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli. Data transformations enabling loop vectorization on multithreaded data parallel architectures. In *ACM SIGPLAN Notices*, volume 45, pages 353–354. ACM, 2010.

[88] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 411–420, 2006.

[89] T. E. Jeremiassen and S. J. Eggers. *Reducing false sharing on shared memory multiprocessors through compile time data transformations*, volume 30. ACM, 1995.

[90] Z. Ji, J. Zhou, M. Takai, J. Martin, and R. Bagrodia. Optimizing parallel execution of detailed wireless network simulation. In *Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pages 162–169, 2004.

[91] B. Jiang and W. K. Chan. Input-based adaptive randomized test case prioritization: A local beam search approach. *Journal of Systems and Software*, 105:91–106, 2015.

[92] B. Jiang, Z. Zhang, W. K. Chan, T. Tse, and T. Y. Chen. How well does test case prioritization integrate with statistical fault localization? *Information and Software Technology*, 54(7):739–758, 2012.

[93] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, 2009.

[94] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo. Development of autonomous car—part i: Distributed system architecture and development process. *IEEE Transactions on Industrial Electronics*, 61(12):7131–7140, 2014.

[95] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering*, 29(3):195–209, 2003.

[96] W. Jun, Z. Yan, and J. Chen. Test case prioritization technique based on genetic algorithm. In *2011 International Conference on Internet Computing and Information Services*, pages 173–175. IEEE, 2011.

[97] A. Kalaji, R. M. Hierons, and S. Swift. A testability transformation approach for state-based programs. In *2009 1st International Symposium on Search Based Software Engineering*, pages 85–88. IEEE, 2009.

[98] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 285–297. IEEE Computer Society Press, 1998.

[99] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, 1999.

[100] S. Kappler. Finding and breaking test dependencies to speed up test execution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1136–1138, 2016.

[101] A. Kaur and S. Goyal. A genetic algorithm for fault based regression test case prioritization. *International Journal of Computer Applications*, 32(8):975–8887, 2011.

[102] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh. An improved method for test case prioritization by incorporating historical test case data. *Science of Computer Programming*, 78(1):93–116, 2012.

[103] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th international conference on software engineering*, pages 119–129, 2002.

[104] T. Kisuki, P. M. Knijnenburg, M. F. O'Boyle, F. Bodin, and H. A. Wijshoff. A feasibility study in iterative compilation. In *International Symposium on High Performance Computing*, pages 121–132. Springer, 1999.

[105] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10–pp. IEEE, 2005.

[106] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 34–43, 2007.

[107] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. *ACM SIGPLAN Notices*, 36(5):156–167, 2001.

[108] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001.

[109] R. Krishnamoorthi and S. S. A. Mary. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology*, 51(4):799–808, 2009.

[110] P. Kukol. System and methods for optimizing object-oriented compilations, Jan. 2 1996. US Patent 5,481,708.

[111] A. Kumar. Development at the speed and scale of Google. *QCon San Francisco*, 2010.

[112] M. Kumar, A. Sharma, and R. Kumar. Fuzzy entropy-based framework for multi-faceted test case classification and selection: an empirical study. *IET software*, 8(3):103–112, 2013.

[113] M. Kumar, A. Sharma, and R. Kumar. An empirical evaluation of a three-tier conduit framework for multifaceted test case classification and selection using fuzzy-ant colony optimisation approach. *Software: Practice and Experience*, 45(7):949–971, 2015.

[114] C. Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[115] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[116] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using hard macros to reduce FPGA compilation time. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 438–441. IEEE, 2010.

[117] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 81–91. IEEE, 2009.

[118] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

[119] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 442–453, 2003.

[120] W. K. Lewchuk. Prefetching data using profile of cache misses from earlier code executions, Apr. 4 2000. US Patent 6,047,363.

[121] B. Li, D. Qiu, H. Leung, and D. Wang. Automatic test case selection for regression testing of composite service based on extensible bpel flow graph. *Journal of Systems and Software*, 85(6):1300–1324, 2012.

[122] N. Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 220–229. IEEE, 2009.

[123] Y. Li and G. Fraser. Bytecode testability transformation. In *International Symposium on Search Based Software Engineering*, pages 237–251. Springer, 2011.

[124] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[125] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4):225–237, 2007.

[126] C. Lin, C. Chen, C. Tsai, and G. M. Kapfhammer. History-based test case prioritization with software version awareness. In *2013 18th International Conference on Engineering of Complex Computer Systems*, pages 171–172, 2013.

[127] Y. Lou, D. Hao, and L. Zhang. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 46–57. IEEE, 2015.

[128] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN notices*, volume 40, pages 190–200. ACM, 2005.

[129] T. Ma, H. Zeng, and X. Wang. Test case prioritization based on requirement correlations. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 419–424, 2016.

[130] R. U. Maheswari and D. J. Mala. Combined genetic and simulated annealingapproach for test case prioritization. *Indian Journal of Science and Technology*, 8(35):1, 2015.

[131] N. Mansour, H. Takkoush, and A. Nehme. Uml-based regression testing for oo software. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(1):51–68, 2011.

[132] C. Mao and Y. Lu. Regression testing for component-based software systems by enhancing change information. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 8–pp. IEEE, 2005.

[133] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543. IEEE, 2013.

[134] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[135] P. McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1689–1696, 2009.

[136] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(3):1–27, 2009.

[137] L. Mei, Y. Cai, C. Jia, B. Jiang, W. K. Chan, Z. Zhang, and T. H. Tse. A subsumption hierarchy of test case prioritization for composite services. *IEEE Transactions on Services Computing*, 8(5):658–673, 2015.

[138] L. Mei, W. K. Chan, T. Tse, B. Jiang, and K. Zhai. Preemptive regression testingof workflow-based web services. *IEEE Transactions on Services Computing*, 8(5):740–754, 2014.

[139] A. Memon, A. Nagarajan, and Q. Xie. Automating regression testing for evolving gui software. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):27–64, 2005.

[140] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):1–36, 2008.

[141] A. M. Memon and M. L. Soffa. Regression testing of guis. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127, 2003.

[142] T. Miller et al. Using dependency structures for prioritization of functional test suites. *IEEE transactions on software engineering*, 39(2):258–275, 2013.

[143] B. Miranda and A. Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, 131:528–549, 2017.

[144] S. Mirarab, S. Akhlaghi, and L. Tahvildari. Size-constrained regression test case selection using multicriteria optimization. *IEEE transactions on Software Engineering*, 38(4):936–956, 2011.

[145] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *International Conference on Fundamental Approaches to Software Engineering*, pages 276–290. Springer, 2007.

[146] S. Mirarab and L. Tahvildari. An empirical study on bayesian network-based approach for test case prioritization. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 278–287. IEEE, 2008.

[147] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with korat. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 135–144, 2007.

[148] M. Muja and D. G. Lowe. Fast matching of binary features. In *Computer and Robot Vision (CRV), 2012 Ninth Conference on*, pages 404–410. IEEE, 2012.

[149] T. Muthusamy and K. Seetharaman. A new effective test case prioritization for regression testing based on prioritization algorithm. *Int. J. Appl. Inf. Syst.(IJAIS)*, 6(7):21–26, 2014.

[150] R. Nagar, A. Kumar, S. Kumar, and A. S. Baghel. Implementing test case selection and reduction techniques using meta-heuristics. In *2014 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)*, pages 837–842. IEEE, 2014.

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

[152] E. N. Narciso, M. E. Delamaro, and F. D. L. D. S. Nunes. Test case selection: A systematic literature review. *International Journal of Software Engineering and Knowledge Engineering*, 24(04):653–676, 2014.

[153] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN notices*, volume 42, pages 89–100. ACM, 2007.

[154] S. Nidhra and J. Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[155] T. Noguchi, H. Washizaki, Y. Fukazawa, A. Sato, and K. Ota. History-based test case prioritization for black box testing using ant colony optimization. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–2. IEEE, 2015.

[156] M. F. O'Boyle and P. M. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *International Journal of Parallel Programming*, 27(3):131–159, 1999.

[157] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontent to support the regression testing of component-based software. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 716–725. IEEE, 2001.

[158] Y. Pang, X. Xue, and A. S. Namin. Identifying effective test cases through k-means clustering for enhancing regression testing. In *2013 12th International*

Conference on Machine Learning and Applications, volume 2, pages 78–83. IEEE, 2013.

[159] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, 41(4):358–383, 2014.

[160] V. Pieterse, D. G. Kourie, L. Cleophas, and B. W. Watson. Performance of C++ bit-vector implementations. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 242–250. ACM, 2010.

[161] J. Poovey, M. Levy, S. Gal-On, and T. Conte. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, PP(99):1–1, 2009.

[162] C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee. Reference distance as a metric for data locality. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, pages 151–156. IEEE, 1997.

[163] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189, 2013.

[164] A. Rajan. *Coverage metrics for requirements-based testing*. PhD thesis, University of Minnesota, 2009.

[165] A. Rajan, S. Sharma, P. Schrammel, and D. Kroening. Accelerated test execution using GPUs. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 97–102. ACM, 2014.

[166] A. Richards. Codeplay software ltd., 2002. "https://www.codeplay.com/".

[167] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Notices*, volume 33, pages 38–49. ACM, 1998.

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

[169] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Software Testing Verification and Reliability*, 10(2):77–109, 2000.

[170] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, 1999.

[171] S. Sabharwal, R. Sibal, and C. Sharma. Prioritization of test case scenarios derived from activity diagram using genetic algorithm. In *2010 International Conference on Computer and Communication Technology (ICCCT)*, pages 481–485, 2010.

[172] A. Sajeev and B. Wibowo. Regression test selection based on version changes of components. In *Tenth Asia-Pacific Software Engineering Conference, 2003.*, pages 78–85. IEEE, 2003.

[173] S. Sampath, R. Bryce, and A. M. Memon. A uniform representation of hybrid criteria for regression testing. *IEEE Transactions on Software Engineering*, 39(10):1326–1344, 2013.

[174] F. Schneider and B. Berenbach. A literature survey on international standards for systems requirements engineering. *Procedia Computer Science*, 16:796–805, 2013.

[175] A. Sen. A quick introduction to the Google C++ testing framework. *IBM DeveloperWorks*, page 20, 2010.

[176] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim. A comparative study on automated software test oracle methods. In *2009 Fourth International Conference on Software Engineering Advances*, pages 140–145. IEEE, 2009.

[177] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008.

[178] K. Solanki, Y. Singh, S. Dalal, and P. R. Srivastava. Test case prioritization: An approach based on modified ant colony optimization. In *Emerging Research in Computing, Information, Communication and Applications*, pages 213–223. Springer, 2016.

[179] H. Srikanth, M. Cashman, and M. B. Cohen. Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. *Journal of Systems and Software*, 119:122–135, 2016.

[180] H. Srikanth, C. Hettiarachchi, and H. Do. Requirements based test prioritization using risk factors: An industrial study. *Information and Software Technology*, 69:71–83, 2016.

[181] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–, 2005.

[182] H. Stallbaum, A. Metzger, and K. Pohl. An automated technique for risk-based test case generation and prioritization. In *Proceedings of the 3rd international workshop on Automation of software test*, pages 67–70, 2008.

[183] R. Stallman, R. Pesch, S. Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.

[184] R. M. Stallman et al. *Using and porting the GNU compiler collection*, volume 86. Free Software Foundation, 1999.

[185] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[186] P. Stratis. Improving test execution time with improved cache locality. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 82–84. IEEE, 2017.

[187] P. Stratis and G. Brown. Assessing the effect of device-based test scheduling on heterogeneous test suite execution. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 193–198, 2018.

[188] P. Stratis and A. Rajan. Test case permutation to improve execution time. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 45–50. IEEE, 2016.

[189] P. Stratis and A. Rajan. Reordering tests for faster test suite execution. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 442–443, 2018.

[190] P. Stratis and A. Rajan. Speeding up test execution with increased cache locality. *Software Testing, Verification and Reliability*, 28(5):e1671, 2018.

[191] P. Stratis, V. Yaneva, and A. Rajan. Assessing the effect of data transformations on test suite compilation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[192] B. Stroustrup. The design of C++ 0x. *C/C++ Users Journal*, 23(5):7, 2005.

[193] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 41–50, 2014.

[194] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark, and S. Larsson. Towards earlier fault detection by value-driven prioritization of test cases using fuzzy topsis. In *Information Technology: New Generations*, pages 745–759. Springer, 2016.

[195] C. Tao, B. Li, X. Sun, and C. Zhang. An approach to regression test selection based on hierarchical slicing technique. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 347–352. IEEE, 2010.

[196] C. Tao, B. Li, X. Sun, and Y. Zhou. A hierarchical model for regression test selection and cost analysis of java programs. In *2010 Asia Pacific Software Engineering Conference*, pages 290–299. IEEE, 2010.

[197] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.

[198] N. Tillmann and W. Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.

[199] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328, Jan 1996.

[200] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 204–215. IEEE, 2003.

[201] W.-T. Tsai, X. Zhou, R. A. Paul, Y. Chen, and X. Bai. A coverage relationship model for test case selection and ranking for multi-version software. In *High Assurance Services Computing*, pages 285–311. Springer, 2009.

[202] E. Ufuktepe and T. Tuglular. Automation architecture for bayesian network based test case prioritization and execution. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 52–57. IEEE, 2016.

[203] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 95–106, June 2000.

[204] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Reliability, quality and safety of software-intensive systems*, pages 3–21. Springer, 1997.

[205] Y. Wang, X. Zhao, and X. Ding. An effective test case prioritization method based on fault severity. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 737–741. IEEE, 2015.

[206] L. J. White and H. K. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings Conference on Software Maintenance 1992*, pages 262–271. IEEE, 1992.

[207] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys (CSUR)*, 30(4):459–527, 1998.

[208] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Notices*, volume 26, pages 30–44. ACM, 1991.

[209] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[210] Y. Wu, M.-H. Chen, and H. M. Kao. Regression testing on object-oriented programs. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*, pages 270–279. IEEE, 1999.

[211] G. Xu and A. Rountev. Regression test selection for aspectj software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 65–74. IEEE, 2007.

[212] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression testing for web applications based on slicing. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pages 652–656. IEEE, 2003.

[213] Z. Xu, K. Gao, T. M. Khoshgoftaar, and N. Seliya. System regression test planning with a fuzzy expert system. *Information Sciences*, 259:532–543, 2014.

[214] Z. Xu, Y. Liu, and K. Gao. A novel fuzzy classification to enhance software regression testing. In *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 53–58. IEEE, 2013.

[215] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[216] H. Yoon and B. Choi. A test case prioritization based on degree of risk exposure and its empirical study. *International Journal of Software Engineering and Knowledge Engineering*, 21(02):191–209, 2011.

[217] M. Yoon, E. Lee, M. Song, B. Choi, et al. A test case prioritization through correlation of requirement and risk. *Journal of Software Engineering and Applications*, 5(10):823, 2012.

[218] M. Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[219] L. Yu, L. Xu, and W.-T. Tsai. Time-constrained test selection for regression testing. In *International Conference on Advanced Data Mining and Applications*, pages 221–232. Springer, 2010.

[220] T. Yu, X. Qu, M. Acharya, and G. Rothermel. Oracle-based regression test selection. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 292–301. IEEE, 2013.

[221] Y. T. Yu and M. F. Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2):179–202, 2012.

[222] F. Yuan, Y. Bian, Z. Li, and R. Zhao. Epistatic genetic algorithm for test case prioritization. In *International Symposium on Search Based Software Engineering*, pages 109–124. Springer, 2015.

[223] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 192–201. IEEE, 2013.

[224] X. Zhao, Z. Wang, X. Fan, and Z. Wang. A clustering-bayesian network based approach for test case prioritization. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 542–547. IEEE, 2015.

[225] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating boolean satisfiability with configurable hardware. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 186–195. IEEE, 1998.

[226] W. Zou, S. M. Reddy, I. Pomeranz, and Y. Huang. SOC test scheduling using simulated annealing. In *VLSI Test Symposium, 2003. Proceedings*, pages 325–330. IEEE, 2003.