Assessing the Effect of Data Transformations on Test Suite Compilation

Panagiotis Stratis
School of Informatics
University of Edinburgh, UK
s1329012@sms.ed.ac.uk

Vanya Yaneva
School of Informatics
University of Edinburgh, UK
vanya.yaneva@ed.ac.uk

Ajitha Rajan
School of Informatics
University of Edinburgh, UK
arajan@ed.ac.uk

ABSTRACT

Background. The requirements and responsibilities assumed by software has 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 tests. Large test suites result in a corresponding increase in the size of the test code that sets up, exercises and verifies the tests. Time needed to compile and optimise the test code becomes prohibitive for large test code sizes.
Aims. In this paper we demonstrate for the first time *optimisations to speedup compilation of test code*. Reducing the compilation time of test code for large and complex systems will allow additional tests to be compiled and executed, while also enabling more frequent and rigorous testing.
Methods. We propose transformations that reduce 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.
Results. Our approach resulted in significant compilation speedups in the range of 1.3× to 69×. Execution of the test code was just as fast with our transformation when compared to the original while also preserving correctness of execution. Finally, our experiments show that the gains in compilation time allow significantly more tests to be included in a single binary, improving scalability of test code compilation.
Conclusions. The proposed transformation results in faster test code compilation for all the programs in our experiment, with more significant speedups for larger case studies and larger numbers of tests. As systems get more complex requiring frequent and extensive testing, we believe our approach provides a safe and efficient means of compiling test code.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; **Compilers**; **Software testing and debugging**;

KEYWORDS

Data Transformations, Testing, Compilers

ACM Reference Format:
Panagiotis Stratis, Vanya Yaneva, and Ajitha Rajan. 2018. Assessing the Effect of Data Transformations on Test Suite Compilation. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239235.3240499

ESEM '18, October 11–12, 2018, Oulu, Finland

ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3240499

1 INTRODUCTION

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 [30, 31]. 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 [2, 10, 14].

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 [23]. Microsoft report that testing code changes is time consuming and annual cost of regression testing exceeds tens of millions of dollars [13]. Codeplay Software [32], who develop specialised tools, including compilers, runtimes and debuggers for heterogenous sytems, use CI for their development, which necessitates frequent compilation and running of large numbers of tests, taking huge amounts of time.

Large test code size. Developing and maintaining the large number of tests needed when testing industrial software is facilitated by test tools and frameworks. *Test code* is a piece of code written in a regular programming language, like Java or C++, that is executed on the system under test (SUT) for the purpose of testing the SUT and observing its behaviour. According to Yousifouglu et al. [40], test code for a given test usually includes four distinct steps that are executed in sequence:

(1) *Setup:* Setup the test fixture with the desired inputs, environment values, and a means to observe the actual outcome.
(2) *Exercise:* Exercise the SUT with the test.
(3) *Verify:* Determine if the test passes or fails by checking if the actual outputs from the SUT match expectations.
(4) *Teardown:* Tear down the test fixture and restore the state of the SUT and environment.

With increasing numbers of tests, the size of the test code becomes very large and according to Microsoft [28], *"A tremendous amount of coding effort goes into writing test code"*. This is often encountered in TDD and CI developments, where test code can be larger than program source code [2].

Large test code sizes 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 are known for its long compilation times [22]. 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.

1.1 Contributions

In this paper, we present a novel approach to speedup compilation of test code and empirically assess its benefits. We propose transformations that restructure test inputs and reduce the number of calls to functions being tested in the test code. Number of instructions in test code reduces significantly with these transformations. We empirically evaluate the effect of the proposed data transformations on test code compilation using two popular C compilers - GCC [11] and Clang [24], enabling all their optimisations. We used industry standard benchmarks – applications from the automotive and telecom domains of the EEMBC benchmark suite [29] for embedded systems, and compute intensive performance benchmarks from SPEC [7]. We also used an industrial application developed by Codeplay Software – ComputeCPP [6] enables acceleration of C++ applications on heterogeneous compute systems using the SYCL [12] 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 transformations 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 transformations 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 transformations 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 paper is organized as follows. Section 2 discusses background and related work. Section 3 presents our approach for reducing compilation time of test code. Our experimental methodology is described in Section 4. Section 5 presents the results from our experiments. Section 6 discusses the threats to validity in our experiment and finally, Section 7 concludes.

2 BACKGROUND AND RELATED WORK

Compiler optimizations [1] 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. 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 [37]. Furthermore, the GNU compiler collection [35] 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. [20] 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 [21] 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. [25] propose the use of pre-compiled circuit blocks, known as *hard macros*, as a way to speed up the compilation process. Chan et al. [5] 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 [41].
- Machine learning techniques have also been proposed for reducing compilation time. Cavazos and O'Boyle [4] 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. [26] 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 [19] and evaluated by Fursin et al. in [9], 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 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 [9], [3] and [39] 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.

In particular, we apply *data transformation* to achieve reduction in test code compilation time. Data transformations are defined by Boyle et al. [27] as "those transformations concerned with the layout, storage and access of array data, rather than reordering the program control flow". In this work, 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. [18], [17] and [33] utilize data transformations in order to improve cache memory locality. In [16], data transformations are used for reducing the number of false sharing misses in a shared memory multiprocessing system and in [15], 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.

Our work, in this paper, is applied to *parametrised unit tests (PUTs)*, introduced by Tillman and Schulte in [38] and also used in commercial test frameworks like GoogleTest [34]. 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 APPROACH

A typical test in a test code, as described in Section 1, 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. Figure 1 shows a test code sample from GoogleTest [34], a popular C++ framework for test code development and execution. There are two test groups in Figure 1, also referred to as parameterized test suites in GoogleTest. 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.

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 groups. This is illustrated in Figure 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 the compiler operates.

3.1 Test Code Transformation

Algorithm 1 illustrates the steps in our transformation. It takes two inputs – the test code (*TC*) and the name of the program function

```
TEST_P(PrimeTableTestSmpl7, ReturnsTrueForPrimes) {
  EXPECT_TRUE(table_->IsPrime(2));
  EXPECT_TRUE(table_->IsPrime(3));
  EXPECT_TRUE(table_->IsPrime(5));
  EXPECT_TRUE(table_->IsPrime(7));
  EXPECT_TRUE(table_->IsPrime(11));
  EXPECT_TRUE(table_->IsPrime(131));
}

TEST_P(PrimeTableTestSmpl7, CanGetNextPrime) {
  EXPECT_EQ(2, table_->GetNextPrime(0));
  EXPECT_EQ(3, table_->GetNextPrime(2));
  EXPECT_EQ(5, table_->GetNextPrime(3));
  EXPECT_EQ(7, table_->GetNextPrime(5));
  EXPECT_EQ(11, table_->GetNextPrime(7));
  EXPECT_EQ(131, table_->GetNextPrime(128));
}
```

Figure 1: GoogleTest sample code illustrating parameterized test suites.

(*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 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 every 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.

3.2 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.

4 EXPERIMENT

We evaluate the effectiveness of the transformation proposed in Section 3 using programs from industry standard benchmark families and an industrial application from Codeplay. We seek to investigate the following questions regarding performance and correctness:

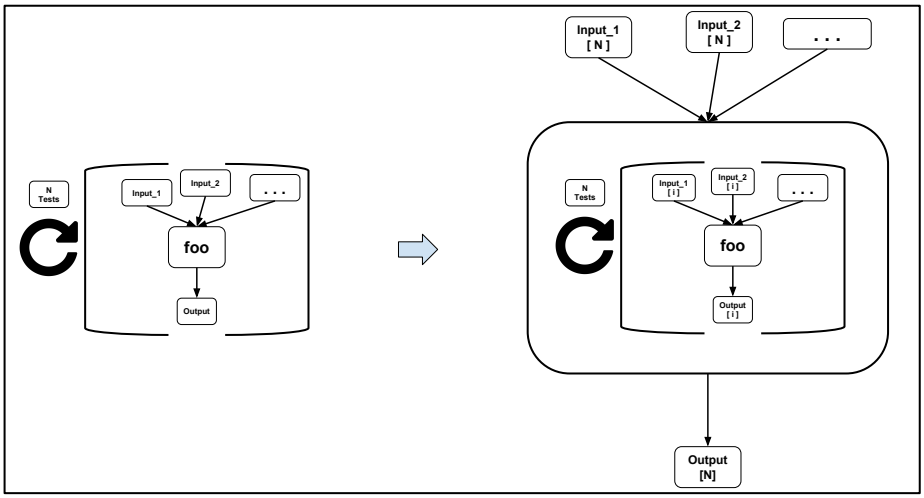

Figure 2: Original test code with N FUT calls (left), transformed to an equivalent test code containing a single FUT call within a loop (right), using our approach.

```
int inputArray0 [] = {0,1,2,3,4,5,6,7,8,9};
int inputArray1 [] = {0,9,2,0,9,1,3,0,0,8};
int inputArray2 [] = {5,8,1,1,5,6,8,2,9,4};
int inputArray3 [] = {9,5,7,7,7,8,5,3,5,0};
int inputArray4 [] = {0,3,5,5,7,1,0,6,7,3};
int inputArray5 [] = {4,2,6,4,2,5,7,3,3,1};
int inputArray6 [] = {3,3,0,4,5,7,4,0,2,2};
int inputArray7 [] = {3,7,4,5,0,8,7,6,0,9};
int inputArray8 [] = {6,1,0,9,1,4,7,1,9,0};
int inputArray9 [] = {0,3,4,5,3,8,1,0,7,3};

int foo(int inputArray [], int inputScalar){
  //Function under test.
}

void TestRunner(){
  ASSERT_EQUALS(expectedValue0, foo(inputArray0, 0));
  ASSERT_EQUALS(expectedValue1, foo(inputArray1, 1));
  ASSERT_EQUALS(expectedValue2, foo(inputArray2, 2));
  ASSERT_EQUALS(expectedValue3, foo(inputArray3, 3));
  ASSERT_EQUALS(expectedValue4, foo(inputArray4, 4));
  ASSERT_EQUALS(expectedValue5, foo(inputArray5, 5));
  ASSERT_EQUALS(expectedValue6, foo(inputArray6, 6));
  ASSERT_EQUALS(expectedValue7, foo(inputArray7, 7));
  ASSERT_EQUALS(expectedValue8, foo(inputArray8, 8));
  ASSERT_EQUALS(expectedValue9, foo(inputArray9, 9));
}
```

```
const int NUM_TESTS= 10;

int inputArray [NUM_TESTS] [ ] = {
{0,1,2,3,4,5,6,7,8,9},
{0,9,2,0,9,1,3,0,0,8},
{5,8,1,1,5,6,8,2,9,4},
{9,5,7,7,7,8,5,3,5,0},
{0,3,5,5,7,1,0,6,7,3},
{4,2,6,4,2,5,7,3,3,1},
{3,3,0,4,5,7,4,0,2,2},
{3,7,4,5,0,8,7,6,0,9},
{6,1,0,9,1,4,7,1,9,0},
{0,3,4,5,3,8,1,0,7,3}
};

int inputScalar [NUM_TESTS] = {0,1,2,3,4,5,6,7,8,9};

int foo(int inputArray [], int inputScalar){
  //Function under test.
  //Remains unchanged after transformation.
}

void TestRunner(){
  for (int i=0; i<NUM_TESTS;i++){
     ASSERT_EQUALS(expectedValue[i], foo(inputArray[i], inputScalar[i]));
  }
}
```

Figure 3: Example of test code transformation.

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.

4.1 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

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* and its input arguments.
3: Merge the input data of the same type from all the tests into centralized data structures *DS*.
4: 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.
5: Update the *PF* call within the loop so that it accepts the correct slice of data from *DS*.
6: Return *TTC*.

Algorithm 1: Test code transformation

range of applications. We also evaluate our approach using an industry provided program, ComputeCPP, developed at Codeplay. 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. The programs and their descriptions along with number of tests are provided in Table 1.

EEMBC. We used 10 subject programs from the Embedded Microprocessor Benchmark Consortium (EEMBC) [29] 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 [8]. 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) [7] 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 bio-molecular 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 [12] standard. SYCL is a single-source C++ programming model for OpenCL [36] 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 combinatory 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 [2], Codeplay 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 developers.

4.2 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 [35] and Clang 5.0 [24], 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*).

5 RESULTS AND ANALYSIS

For each of the subject programs presented in Section 4, 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.

5.1 Q1. Compilation

Figures 4 and 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 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

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 1: Subject programs used in our experiment

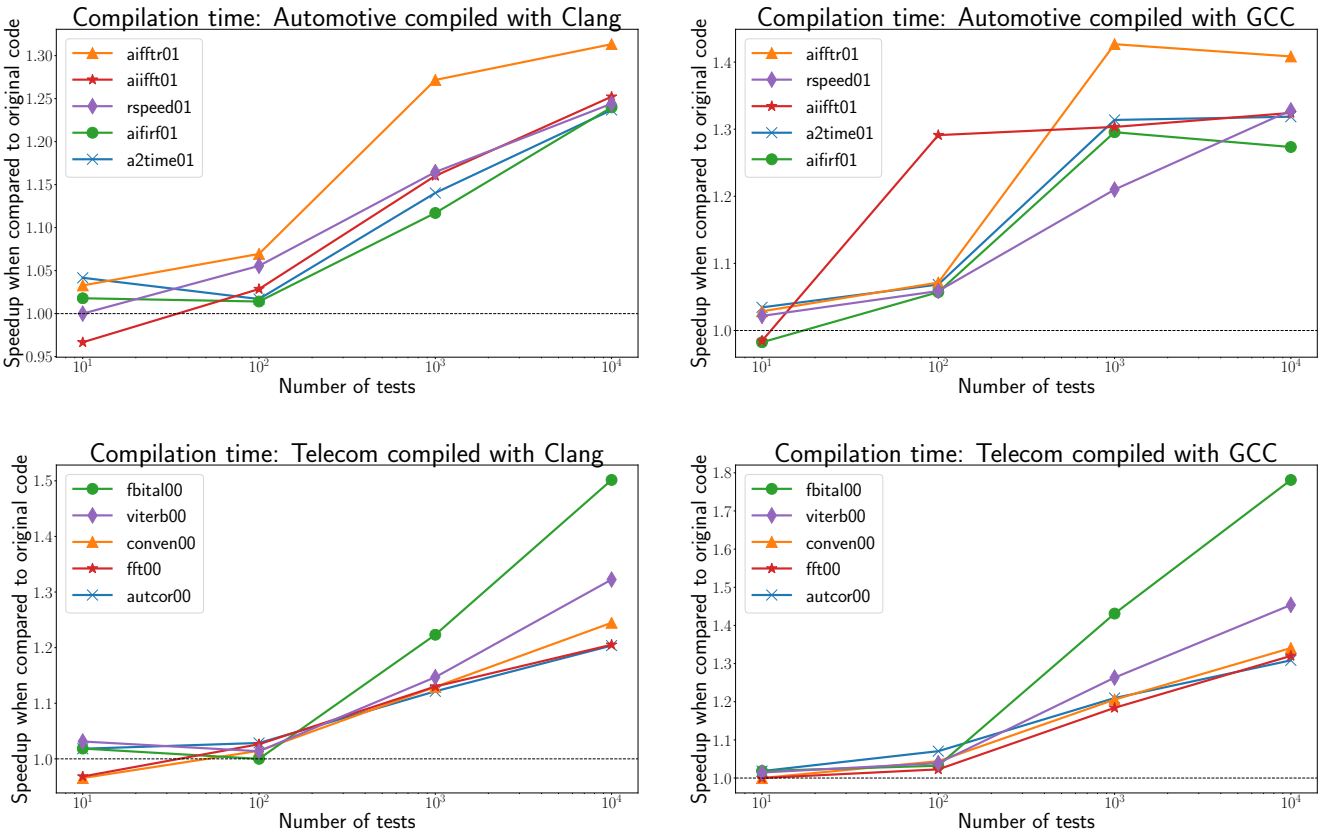

Figure 4: Speedup in compilation time for EEMBC, when compared to the original code, for different test suite sizes.

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.

5.1.1 EEMBC: Automotive and Telecom. The results for EEMBC programs in Figure 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.

5.1.2 SPEC. Figure 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 5.1.5

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 5.1.5.

5.1.3 ComputeCpp. Figure 6 shows the compilation speedup achieved for the two ComputeCPP test suites – bufferTS and imageTS. Original compilation times are shown in Table 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 5.1.4.

5.1.4 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

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.

5.1.5 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 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

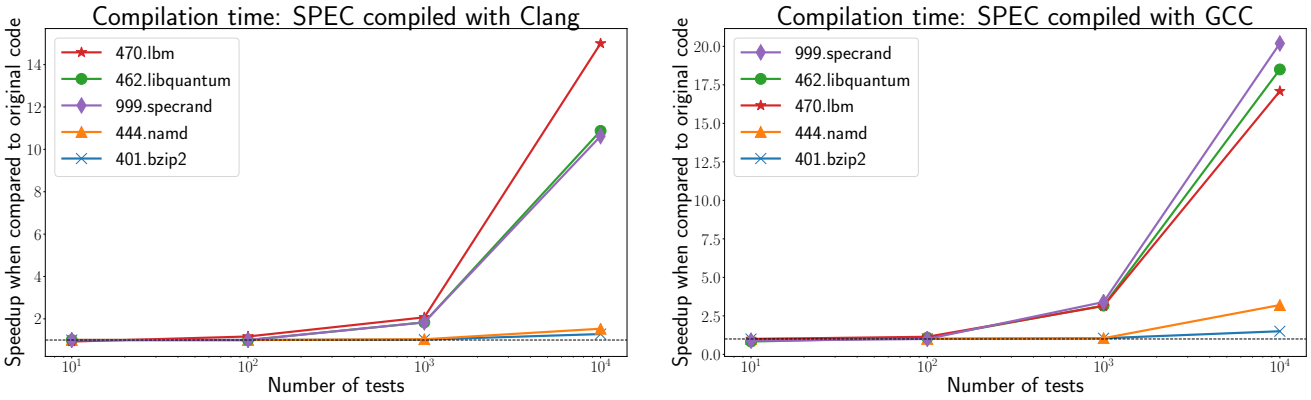

Figure 5: Speedup in compilation time for SPEC, when compared to the original code, for different test suite sizes.

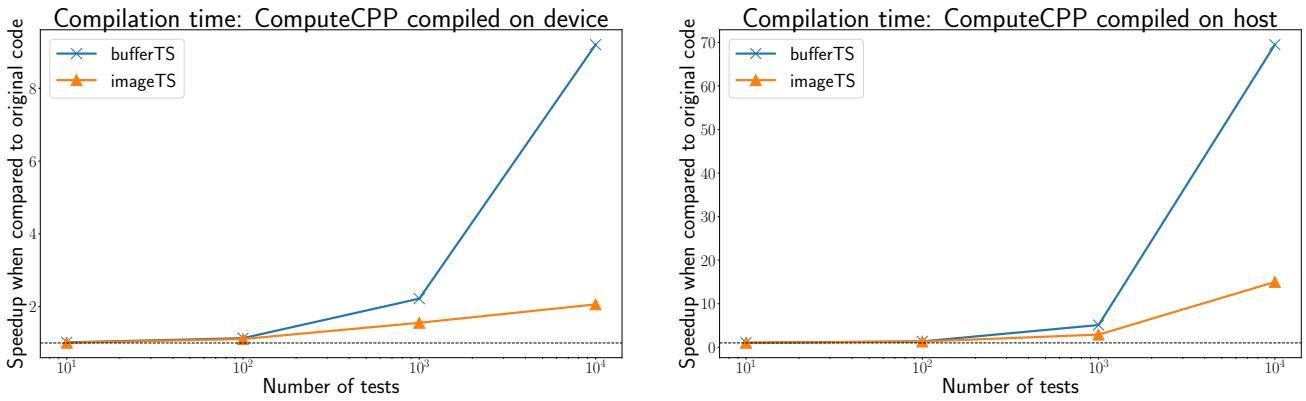

Figure 6: Speedup in compilation time for ComputeCPP, when compared to the original code, for different test suite sizes.

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.

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× to 69×. However, with the EEMBC benchmarks and two of the SPEC benchmarks, our approach achieves very low speedup (less than 2×). 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.

5.1.6 Statistical Analysis. We analyse the results presented in Figures 4,5, 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.

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, larger the number of tests in the test code, 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.

5.2 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 all EEMBC and SPEC programs, we find that the execution of the transformed test code is as fast as the original code. For ComputeCPP, transformed test code executed faster than the original version. For the programs in our experiment, the results categorically show that our transformation does *not* slow down the execution of the test code.

5.3 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.

5.4 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.

6 THREATS TO VALIDITY

We see two 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 runtime, 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 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.

7 CONCLUSION

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 transformations 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

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 2: Compilation times for ComputeCPP test codes.

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, larger the number of tests, 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 paper, we have sampled programs from embedded systems, performance benchmarks, and an industrial application. In the future, we plan to conduct more extensive empirical evaluations using programs and test suites from different domains. We will also extend our approach to handle complex test inputs, like images and files.

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.

[2] Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley.

[3] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*.

[4] John Cavazos and Michael FP O'boyle. 2006. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices* 41, 10 (2006), 229–240.

[5] Pak K Chan, Mark J Boyd, S Goren, K Klenk, V Kodavati, R Kundu, M Margolese, J Sun, K Suzuki, E Thorne, et al. 1999. 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*. IEEE, 308–309.

[6] ComputeCpp. 2017. ComputeCpp – Accelerate Complex C++ Applications on Heterogeneous Compute Systems using Open Standards. "https://www.codeplay.com/products/computesuite/computecpp".

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

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

[9] GG Fursin, Michael FP O'Boyle, and Peter MW Knijnenburg. 2002. Evaluating iterative compilation. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 362–376.

[10] Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2016. 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 (2016), 25.

[11] Brian Gough and Richard M Stallman. 2004. An Introduction to GCC for the GNU Compilers gcc and g++. *Network Theory Ltd* (2004), 35–46.

[12] Khronos OpenCL Working Group et al. 2015. SYCL: C++ Single-source Heterogeneous Programming For openCL.

[13] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *Proceedings of the 37th ICSE*. IEEE Press, 483–493.

[14] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education.

[15] Byunghyun Jang, Perhaad Mistry, Dana Schaa, Rodrigo Dominguez, and David Kaeli. 2010. Data transformations enabling loop vectorization on multithreaded data parallel architectures. In *ACM Sigplan Notices*, Vol. 45. ACM, 353–354.

[16] Tor E Jeremiassen and Susan J Eggers. 1995. *Reducing false sharing on shared memory multiprocessors through compile time data transformations*. Vol. 30. ACM.

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

[18] Mahmut Kandemir, J Ramanujam, and Alok Choudhary. 1999. Improving cache locality by a combination of loop and data transformations. *IEEE Trans. Comput.* 48, 2 (1999), 159–167.

[19] Toru Kisuki, Peter MW Knijnenburg, Mike FP O'Boyle, François Bodin, and Harry AG Wijshoff. 1999. A feasibility study in iterative compilation. In *International Symposium on High Performance Computing*. Springer, 121–132.

[20] Chandra Krintz and Brad Calder. 2001. Using annotations to reduce dynamic optimization time. *ACM Sigplan Notices* 36, 5 (2001), 156–167.

[21] Chandra J Krintz, David Grove, Vivek Sarkar, and Brad Calder. 2001. Reducing the overhead of dynamic compilation. *Software: Practice and Experience* 31, 8 (2001), 717–738.

[22] Peter Kukol. 1996. System and methods for optimizing object-oriented compilations. US Patent 5,481,708.

[23] Ashish Kumar. 2010. Development at the speed and scale of google. *QCon San Francisco* (2010).

[24] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

[25] Christopher Lavin, Marc Padilla, Subhrashankha Ghosh, Brent Nelson, Brad Hutchings, and Michael Wirthlin. 2010. Using hard macros to reduce FPGA compilation time. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 438–441.

[26] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 81–91.

[27] Michael FP O'boyle and Peter MW Knijnenburg. 1999. Nonsingular data transformations: Definition, validity, and applications. *International Journal of Parallel Programming* 27, 3 (1999), 131–159.

[28] Alan Page, Ken Johnston, and Bj Rollison. 2008. *How we test software at Microsoft*. Microsoft Press.

[29] J.A. Poovey, M Levy, S Gal-On, and T Conte. 2009. A Benchmark Characterization of the EEMBC Benchmark Suite. *Micro, IEEE* PP, 99 (2009), 1–1. https://doi.org/10.1109/MM.2009.50

[30] Ajitha Rajan. 2009. *Coverage metrics for requirements-based testing*. Ph.D. Dissertation. University of Minnesota.

[31] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. 2014. Accelerated test execution using GPUs. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 97–102.

[32] Andrew Richards. 2002. Codeplay Software Ltd. "https://www.codeplay.com/".

[33] Gabriel Rivera and Chau-Wen Tseng. 1998. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 38–49.

[34] Arpan Sen. 2010. A quick introduction to the Google C++ Testing Framework. *IBM DeveloperWorks* (2010), 20.

[35] Richard M Stallman et al. 1999. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation.

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

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

[38] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 253–262.

[39] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. 2003. Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 204–215.

[40] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. 2015. Software test-code engineering: A systematic mapping. *Information and Software Technology* 58 (2015), 123–147.

[41] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. 1998. Accelerating Boolean satisfiability with configurable hardware. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*. IEEE, 186–195.