Systematic Testing of Hardware Compilers

Testing the DWARV C-to-HDL Compiler

A.J. Rouvoet, E. Schoute and A.B. Booij

Department of Computer Engineering
Delft University of Technology
Mekelweg 4, 2628 CD Delft
{A.J.Rouvoet, E.Schoute, A.B.Booij}@student.tudelft.nl
Abstract

DWARV is a C-to-VHDL compiler which can compile arbitrary C programs for a heterogeneous platform with minimal effort. Depending on the application this will imply faster execution. We give an overview of C-to-VHDL compilers. One important aspect of all software is its reliability, and we aim to assess DWARV’s reliability and the adequacy of its test suite. Based on the identified shortcomings of the test suite, we implement the test result framework Dummy. We explore some of the additional possibilities created by the usage of Dummy, and provide a plan to improve DWARV’s test suite. We conclude with an analysis of the results that show the advantages of the implemented test framework and proposed test plan.
## Contents

1. Background 5
  1.1. Heterogeneous Multicore Platforms ........................................ 5
    1.1.1. Classification ....................................................... 5
    1.1.2. hArtes Toolchain ..................................................... 6
    1.1.3. Other Heterogeneous Platforms ..................................... 7
  1.2. Testing ................................................................. 7
    1.2.1. Testing Granularities .............................................. 8
    1.2.2. Testing Methodologies ............................................. 8
    1.2.3. Test Suite Adequacy Analysis ..................................... 10
  1.3. Compiler Testing ...................................................... 11
    1.3.1. Lexical and Syntax Analysis ...................................... 12
    1.3.2. Analysis of Static Semantics .................................... 12
    1.3.3. Optimizing Transformations ...................................... 13
    1.3.4. Code Generation .................................................... 14
  1.4. Hardware/Software Co-Design ......................................... 14
    1.4.1. Heterogeneous Platform Compilers ................................ 15
    1.4.2. Difficulties in Testing Hardware/Software Co-design ............ 17
  1.5. DWARV C-to-VHDL Compiler ........................................... 17
    1.5.1. Lexical and Syntax Analysis and Analysis of Static Semantics . 17
    1.5.2. Optimizing Transformations ...................................... 18
    1.5.3. Code Generation .................................................... 18

2. Goal of the Project 20
  2.1. Problem Description .................................................. 20
    2.1.1. Systematic Testing ................................................. 20
    2.1.2. Test Result Management .......................................... 21
  2.2. Requirements ........................................................... 22
    2.2.1. Creating a Test Plan for DWARV .................................. 22
    2.2.2. Improving the Test Framework .................................... 23

3. Solution Design and Implementation 25
  3.1. Adequacy Criteria ...................................................... 25
    3.1.1. Functional Requirement Coverage ................................ 25
    3.1.2. Compiler Rule and Statement Coverage ............................ 25
    3.1.3. Language Rule Coverage .......................................... 26
  3.2. Proposed Testing Methodology ....................................... 26
    3.2.1. Language Coverage ................................................ 26
    3.2.2. Model-Based Testing ............................................... 26
    3.2.3. Identification of Missing Test Cases ..................................... 27
    3.2.4. Random Test Case Generation ..................................... 27
1. Background

Further growth in traditional single-core processor architectures is slowing down due to the *power wall*, which is the increasing power cost related to increasing frequency [1]. To face the ever-increasing demand for more computational power, High Performance Computing (HPC) architectures currently feature massively multicore architectures. Heterogeneous computing is the use of different processing cores, resulting in higher performance by combining the strengths of traditional CPUs (sequential processing), with the use of massively parallel computation cores in devices such as GPUs and FPGAs. To combine both software and hardware in one architecture is also known as Reconfigurable Computing (RC) [2].

We will discuss first the general notion of heterogeneous platforms and what DWARV (a C-to-VHDL compiler) is in Section 1.1. Next up is a look at testing methods in general and testing methods used specifically for compilers, in Section 1.2 and Section 1.3 respectively. Finally, the current state of hardware compilers will be discussed in Section 1.4, of which the emphasis is on ANSI-C to VHDL compilers.

1.1. Heterogeneous Multicore Platforms

Specialized coprocessors (e.g. Cell BE SPUs in the Sony Playstation 3) or data-parallel accelerators (e.g. GPGPUs) are already in widespread use. [3] shows that heterogeneity in supercomputers causes a higher performance / power ratio and a method to port traditional homogeneous applications to the platform. Generally though, programming for a heterogeneous platform is harder than for a homogeneous platform, and some applications are more suitable for homogeneous platforms than for heterogeneous platforms. Heterogeneous computing is not a solution to every computational problem, but is currently mainly focused on highly concurrent computing\(^1\) [4].

1.1.1. Classification

In the previous section we have seen some applications for heterogeneous platforms; let us now take a look at how they accelerate programs and what problems they present. Because of the power wall, research is currently focussed on improving computational performance by the use of parallelism. As proposed in Flynn’s taxonomy [5] for computer architectures, there are four levels of parallelism:

- **SISD** Single Instruction Single Data, a sequential computer that uses no parallelism. A traditional uniprocessor in old PCs is a SISD computer.
- **SIMD** Single Instruction Multiple Data, a processor that processes multiple data streams per instruction, such as a vector processor or a GPU.

\(^1\)Concurrent computing is the execution of computational processes that can either be executed sequentially or in parallel.
**MISD** Multiple Instruction Single Data, uncommonly used. Multiple processors work on a single set of data in parallel.

**MIMD** Multiple Instruction Multiple Data, very common in current HPC computers. Multiple processors execute instructions on multiple data sources.

The GPU has long been in use for personal computers. Originally intended to render 2D and 3D scenes, it is now also being used as SIMD processor with a large amount of cores [1] (for example, the consumer NVidia GeForce GTX770 has 1536 cores [6]). Because a CPU controls the GPU, the system is effectively heterogeneous. GPUs are capable of applying a single instruction to a large uniform dataset significantly faster than the CPU.

FPGAs are an upcoming computation platform that provide more processing cores at a lower power cost. They are reprogrammable integrated circuits, which allows them to function as either SIMD or MIMD processors. Compared to a GPU, an FPGA can perform more simple calculations in parallel and provide better pipelining [7]. GPUs, on the other hand, run at higher clock frequencies and have access to a better memory architecture, which supports multiple access methods (e.g. random access, FIFO, stack).

**1.1.2. hArtes Toolchain**

At Delft University of Technology, research is being done into heterogeneous multicore computation which is based on the Molen machine organization [8]. The Molen machine organisation is a shared memory processor co-processor architecture where the co-processors can be either DSPs, GPU based or consist of kernels implemented on FPGAs [9]. Among other things, this allows for mapping functions to hardware, parallel execution and application portability between multiple hardware platforms. In Fig. 1.1 we depict the various components of the Molen machine organization. The Arbiter routes the instruction being executed to the appropriate
component, such as the CCUs and the FPGAs in this case. The Data Memory Mux/Demux unit together with the Data Fetch unit are responsible for unified memory access across system components.

The Delft Workbench [10] was built with these ideas in mind to transform applications to run on reconfigurable hardware components.

A vital component of the Delft Workbench is the hardware compiler which, in principle, allows software engineers to use FPGA based acceleration without the need to develop the hardware kernel themselves. To this purpose, the DWARV compiler was developed which takes C as input and produces synthesizable VHDL [11, 12]. With DWARV significant speedups were observed in hardware execution compared to software execution. Between 1.4x and 6.8x speedup for applications was observed and later up to 4.41x compared to LegUp [13]. For more details see Section 1.3.

The hArtes (Holistic Approach to Reconfigurable Real-Time Embedded Systems) project aims to provide an integrated toolchain for the heterogeneous design process [9]. hArtes is the culmination of previous work and includes components of the Delft Workbench. It includes the Algorithm Exploration Toolbox (AET), the Design Space Exploration Toolbox (DSE) and the Synthesis Toolbox. The AET provides tools for developing algorithms at a high abstraction level using a GUI, which can then be compiled to C. The output of the AET or custom code is then fed to DSE, which performs partitioning and mapping transformations. Finally, the output of DSE is given to the Synthesis Toolbox, a set of C-compilers for each Processing Element (e.g. GPPs, FPGAs, etc.). A team from hArtes continues its development in a derived company called BlueBee Multi-core Technologies.

1.1.3. Other Heterogeneous Platforms

Companies that have heterogeneous platform products include Convey and Xilinx. Convey produces the HC-1 and HC-2 systems, both of which combine an Intel processor with a number of FPGA coprocessors. The Convey HC-2 contains 2 Intel Xeon processors [14] with 4 Xilinx Virtex FPGAs. Convey delivers a HC-1 simulator together with various FORTRAN, C and C++ compilers for development. Xilinx has the Zynq-7000 All Programmable SoC (System on a Chip) which contains an ARM processor combined with an FPGA [15]. Xilinx has also released the Xilinx Vivado Design suite, which capable of C-to-HDL compilation [16] (see also Section 1.4.1).

1.2. Testing

While a wide range of testing methodologies exist such as “regression testing” and “continuous integration” [17], no single methodology assure quality or correctness by itself. Therefore it is of importance to determine how testing can contribute to the quality assurance of the product under development and how it is done correctly and effectively.

In this section we will look at a general definition of testing and how it is often done. In the next section we will discuss compiler testing specifically.
1.2.1. Testing Granularities

Testing can be done on different granularities. Specifically, we distinguish unit testing, integration testing and system testing. The different testing levels serve different purposes, which we will discuss in this section.

Unit Testing

The goal of unit testing is to show that individual functions in a program perform correctly in isolation of the rest [17]. An advantage of this is that the functionality can be tested independently of the state of the program and many edge cases can be covered using white box testing (i.e. testing with insight in program code), even if such a case may not (yet) occur in the context of the program. Additionally, integration is a lot easier, when all individual parts are shown to be functioning correctly.

However, writing tests for each individual function is very time consuming. Depending on available time, testing could be limited to important components. Additionally, because the edge cases may never occur while running the program, one might be doing unnecessary testing [17]. This can be solved by enforcing preconditions for the methods that exclude the edge cases.

Integration Testing

Integration testing is used to test requirements on the component level of the system. Its purpose is to establish that components adhere to the requirements specifications of other components. This means that testing is done without knowledge of the actual implementation, in other words without access to the source code. For integration testing only the software interfaces and their intended functionality have to be known.

System Testing

System testing is another form testing in which the entire system is tested at once [17, Chapter 22]. In the case of DWARV this includes both software and hardware tests. The software tests of DWARV check the functionality on a simulated hardware platform, while the hardware tests check the performance on the heterogeneous platform.

The advantages of system testing is that one is testing the compiler in the context of its real deployment.

1.2.2. Testing Methodologies

Besides the types of tests that can be performed, different methodologies exist that prescribe when and how to test. In this section we will discuss several practices that are relevant for this project.
Regression testing

Regression testing is a testing methodology that is used to uncover bugs in applications after changes are made to them [17, Chapter 22]. One way of regression testing is using unit tests to see if new code breaks existing unit tests. This allows for quick diagnosis of defects and prevents further work with malfunctioning software. Regression testing involves running unit tests on affected code, however, as mentioned earlier, writing unit tests can be time consuming.

Continuous Integration (CI) is a form of regression testing and is designed to prevent what is sometimes referred to as “integration hell”. As explained in [18], while working on code from a version control system (e.g., SVN, Git) repository, both your copy from the code and the origin repository are often changing. This implies that once one is ready to commit work to the repository, the work done on both ends have to be merged. The further apart these branches are, the more conflicts it often results in.

CI attempts to prevent the problems resulting from this, by practices such as merging with the repository every morning and running the unit tests before or after every commit.

CI is often done with an integration server, which runs the configured tests on each commit, and communicates the results to the developers. Additionally, test results are often available on an online dashboard to keep track of the project status.

Test Driven Development

A core technique in several agile development methods (e.g., Extreme Programming [18], Scrum [19]), is Test Driven Development (TDD), which focuses on developing tests before implementing functionality [20]. In the TDD workflow, a developer writes a unit test that tests for adherence to specifications. Then, the developer implements the functionality, and finally checks for correctness of the implementation using the test. This is done in very small development cycles and slowly builds a large number of unit tests which can later on still check for proper functioning after modifications or maintenance.

Model-Based Testing

Using models derived from expected behaviour to produce test case specifications allows for a systematic method of developing tests, called model-based testing [17, Chapter 14]. Tests can be developed from the derived test case specifications that check if the program behaviour is as expected from the model. There are many kinds of models that can be derived from the program specification, some more formal than others (e.g., finite state machines versus class diagrams).

A common use for model-based testing is decision structures. By modelling a decision such as an if statement as a predicate, it is easier to extract a decision table from the statement [17]. An example of such a predicate can be

\[(condition1 \land condition2) \lor \neg condition3,\]  

which simply states that either condition1 and condition2 have to be true, or condition3 has to be false. This is often used in modified condition/decision adequacy criterion (MC/DC), to formulate a minimum number of tests for a decision structure.
Test Case Generation

Many projects are tested using fixed test suites, that is to say, the test cases are written by developers to reflect common cases, edge cases and maybe cases that are/were known to result in faulty behaviour. Testing this way allows the developers to assess if there are changes in such border cases during development. A disadvantage of this method is that it is likely that a lot of cases will never be covered because they are atypical. Even though those cases are atypical, they are not unimportant. As is noted by [21], developers writing embedded system or kernel code or generating code could well make use of these atypical cases.

For systems of which a good specification exists, it is possible to generate test cases automatically. Random, but systematic, test case generation can help the developers cover atypical cases, without failing to meet the coverage criteria.

Many methods have been developed for generating test cases, depending on the available specification. For example, for compilers, a complete grammar is known that fully specifies the inputs that the lexical/syntax analyser should accept. One could use the rules of the grammar to create a minimal set of test inputs that covers all the rules of the grammar (known as Purdom’s algorithm[22]).

1.2.3. Test Suite Adequacy Analysis

Apart from the actual test results it is is important to have a measure of the adequacy of the test suite. Complete test suite adequacy, i.e. a test suite that ensures correctness of the program under test, is not possible [17, Ch. 9.1]. Criteria are used to identify inadequacies in test suites. From every criterion that is not satisfied by the test suite, we can conclude that the test suite does not guard against a specific type of faults in the program.

In this section various methods for evaluating test suite adequacy criteria are discussed.

Figure 1.2.: An example Control Flow Graph (CFG).
**Code Coverage**

One such method of judging test suite adequacy used in structural testing is using code coverage criteria. A program can be expressed as a Control Flow Graph (CFG), of which an example can be found in Fig. 1.2. Using the CFG we can discern certain levels of structural code coverage [17]:

- **Statement testing** is checking if each node in the CFG of a program has been executed by the test suite. Line coverage can be used to perform statement testing.

- **Branch testing** is checking that each branch in the CFG has been taken. Branch coverage is used to perform branch testing.

- **Condition testing** is checking if each branch condition has been tested. A table can be made of all possible combinations of conditions, each of which can then be tested.

- **Path testing** is checking if each possible path through the CFG has been taken. This is typically not realistic however, because of the combinatorial explosion in the amount of paths.

Additionally, there one could distinguish **function point coverage**, which shows what functions have been executed during testing. As such it only gives an indication of where no testing has been done at all.

**Mutation Analysis**

Another adequacy criterion is based on the idea that a test suite should be able to distinguish a correct program from any incorrect program. To test the sensitivity of a test suite to new faults, one seeds faults in a program, through mutations of the original program, to obtain an alternate program with alternate (faulty) behaviour for a subset of program test cases. The adequacy of the test suite is then judged on it’s ability to distinguish the mutant from the original program (1.2) [23].

\[
\text{adequacy} = \frac{\text{non equivalent mutants detected}}{\text{total number of non equivalent mutants}}
\]  

(1.2)

It can be shown that mutation analysis subsumes structural coverage criteria [17, Ch. 16.4]. Although it can thus identify missing test cases that cannot be identified using structural code coverage analysis, it is more difficult to apply in practice.

**1.3. Compiler Testing**

Compilers allow developers to write applications on a higher level of abstraction. Writing in high-level languages such as ANSI-C, Java or Python is considered normal, while writing machine code is viewed as a peculiarity. To abstract having to write a lower level Hardware Description Language (HDL) from developers, the DWARV compiler has been developed.

Compilation can be split into multiple stages [22]:

1. Lexical and syntax analysis,
2. Analysis of static semantics,
3. Optimizing transformations,


Stage 1 verifies that the input code can indeed be generated by the grammar of the language. The second stage will build the parsing tree and validate the static semantics of the program. These are all semantic properties that can be checked without running the program, such as scopes of identifiers and variable types. Programs of which the static semantics are validated, are referred to as “semantically correct programs”.

Once a program is deemed semantically correct, the third stage applies optimizations to the intermediate representation of the code to increase performance. Finally the fourth stage creates the executable file, or whatever the intended output language is.

The importance of a correct compiler is clear: an incorrect compiler would create an executable with behaviour that differs from the original program. Because of its importance and complexity, it is crucial to verify the correctness of a compiler by extensive testing. In the next section, we will examine how compilers are tested based on the model of different compilation stages explained above.

1.3.1. Lexical and Syntax Analysis

Lexical and syntax analysers are tested as a “black box”, i.e. without a priori knowledge of its internals. The grammar of the high level language serves as a complete specification of the acceptable input: the tests are written to validate that the compiler accepts all syntactically correct inputs (positive tests) and rejects any input not adhering to the grammar (negative tests).

Furthermore, because of the generating nature of grammars, a few algorithms exist that generate positive test cases that can ascertain a certain coverage [22]. The authors of [24] provides an implementation of Purdom’s algorithm for the C and C++ languages.

1.3.2. Analysis of Static Semantics

The static semantics of an input file is analyzed using an extension of the context free grammar used in the first stage, called an attribute grammar [22]. Every node has attributes that get their value from semantic functions associated with the grammar rules. These functions express the attribute value in terms of the attribute values of parent nodes. After all attributes of the abstract syntax tree have been calculated, the attribute values are verified using the so called context conditions attached to the grammar rules. If all context conditions evaluate to true, the program is considered statically correct.

As with lexical analysis, several methods exist for generating tests with different coverage criteria. These methods use the attribute grammar (which serves as a complete formal specification of the static semantics) to generate test cases [22].
1.3.3. Optimizing Transformations

As explained, the third stage applies transformations to the intermediate representation generated by the previous stage for optimization purposes. The fourth stage can be seen as a final transformation, from IR to output code. All these transformations are considered correct if they do not alter the observable behavior of the program [22].

To test this, the test program prints the results of the program that will be compiled and the printed value is compared to a reference value. The reference value can be obtained from running the same program, compiled using a different compiler, from a (partial) specification of the dynamic semantics of the language, or a manual evaluation of the attribute grammar [22].

Any transformation that satisfies the above criterion is semantically correct. However, it is important to also verify that the transformation indeed increases performance for (at least) a subset of all programs and does not negatively influence performance in other cases. A common method to establish this is by comparing performance indicators from test runs that respectively include and exclude the transformation.

Systematically design test suites

When designing test suites for testing optimizing transformations, model based testing (see Section 1.2.2) is a practice that can help the developer to perform this systematically. One can apply this method by abstracting the code constructs that the optimizing transformation targets into a model description and then constructing tests based on this model [25].

Using this method, one can create tests to determine if the inputs that will be touched by the transformation are indeed transformed correctly and can formulate tests that cover both positive and negative cases.

Formally Proving Correctness

In [26] and [27] methods are proposed for formally proving the correctness of an optimizing transformation. First, a formal definition of transformation rules is given. A rule is of the form

\[ I \Rightarrow I' \text{ if } \phi, \]  

(1.3)

where \( I, I' \) are intermediate language instructions and \( \phi \) is a property expressed in temporal logic for describing data flow. If a program \( \pi \) contains an instruction of the form \( I \) and flow condition \( \phi \) is satisfied, the rule specifies it is replaced by the instruction \( I' \). To prove correctness of the transformation, it has to be shown that \( \llbracket \pi \rrbracket = \llbracket \pi' \rrbracket \), in other words \( \pi \) is semantically equivalent to the transformed program \( \pi' \). In short, to prove equivalence it must be shown that a sequence of states \( C_1 \) has a simulation relation \( R \) to \( C'_1 \), such that by induction every state from start state \( initial(v) \) the same relation \( R \) holds.

However, formally proving correctness of transformations is time consuming, requires a high degree of expertise and will not be sufficient to meet compiler quality criteria, as the formal proof will only consider the theoretical specification of the transformation and not the discrepancies with the implementation. With a test suite one can determine whether the implementation behaves as specified.
Test Generation

The observable behaviour of a program is determined by the dynamic semantics of the program. Proving that a transformation is correct would consequently have to show that every meaningful program maintains its meaning under the transformation. However, it is impossible to test a compiler for every meaningful program (since this is an infinite set).

If a complete specification of the dynamic semantics exists, test suites could be built based on that specification. A test suite could then be qualified by the coverage of the specification that it provides, similar to the “rule coverage” and “context-dependent rule coverage” qualifications that were formulated for the syntax analyzer test suites. Unlike for the syntax and static semantics however, which are described by the language grammars, a unique formal specification for the dynamic semantics of a program has not been formulated[22]. For real programming languages, a complete specification of dynamic semantics is hardly ever specified. Without such a complete formal specification of the behaviour of the system, it is impossible to generate a test suite that can guarantee a meaningful coverage [25].

Due to this, compiler tests for transformations are mostly hand-coded. Some methods have been developed, however, to generate randomized test cases as an addition to the hand-coded test suites. Although, these randomly generated tests cannot guarantee any meaningful coverage of the dynamic semantics, they are nonetheless very good at discovering bugs in compilers [21].

1.3.4. Code Generation

The code generation stage has the same correctness criterion as any transformation. Model based testing is not useful for this transformation, however, since a model of the input would be a complete description of the dynamic semantics. But since all semantically correct programs that test stages I to III are set up as system tests, they also implicitly test the code generation stage. These tests would cover at least all language constructs. Additionally, a randomly generated test suite could expose many bugs, even with a large test suite, as was demonstrated by the authors of [21].

1.4. Hardware/Software Co-Design

Automatic translation of high level languages (e.g. ANSI C) to hardware will make hardware acceleration easily available for a multitude of problems. Some problems can be significantly accelerated using a accelerator hardware platform, depending on how well the problem can be parallelized. Because hardware platforms offer many more Processing Elements (PEs) than traditional General Purpose Processors (GPPs), they can execute more calculations in parallel. However, the individual PEs are not as fast in sequential execution as a GPP. Whether hardware acceleration is suitable or not depends on the nature of the problem.

Even if a problem is embarrassingly parallelizable, which broadly means it has little to no data sharing between threads and requires little effort for load balancing, implementing it efficiently in hardware still poses an issue [9]. Knowledge of Hardware Description Languages (HDLs) is less readily available than knowledge on higher level languages, such as C, C++, Java. This means that implementation usually requires hardware experts, such as electrical engineers. Because HDLs are lower level languages, a lot more time is necessary to implement the same functionality.
To execute an application on a heterogeneous platform, it first has to be separated in a “software” and a “hardware” section [28]. The software section is compiled for the GPP, while the hardware part is translated to a HDL. The translation to hardware is a non-trivial problem and this section will focus on the different methods that have been developed to this end.

There are various projects that attempt compilation of ANSI-C (which we will now refer to as C) code to VHDL. C is a sequential language, whereas hardware acceleration typically uses concurrency [28]. Hence, for good translation, concurrency in applications will have to be found and exploited automatically.

Some compilers use hardware languages similar to C but are semantically different, e.g. Handel-C [29]. Others extend C with a library that can be used for concurrency and specifying the executing PE, e.g. Streams-C [30]. And a third class, e.g. the Altium designer suite and the Nios II C2H compiler, puts heavy constraints on the C language (such as not allowing the use of the main memory) to keep it synthesizeable as a pure HDL entity. In all these cases, existing application code will have to be rewritten to take advantage of the heterogeneous acceleration. Additionally, engineers working on a problem will need to have knowledge of hardware to effectively translate their application to the new platform.

Another method is compiling (parts of) unmodified C to VHDL with as few as possible constraints. These constraints exist because certain language constructs, such as recursion, do not gain in speed when translated to accelerator hardware, because of their sequential nature. Software can, however, often be modified to meet the constraints (e.g. tail recursion can be rewritten as a loop). We will refer to these compilers as “heterogeneous platform compilers”. In the next section, five heterogeneous platform compilers will be compared.

1.4.1. Heterogeneous Platform Compilers

The goal of HPCs is to compile software to run not only on the CPU core, but rather to run in parallel on the different heterogeneous cores available, in such a way that they maximally utilize the capabilities of these cores. If, for example, an FPGA core is available, the compiler can compile functions suitable for parallelization to be run on the FPGA by compiling the function to a hardware module.

This eliminates the disadvantage of C-to-HDL compilers mentioned in the previous section that prohibits software engineers to utilize the full capabilities of a heterogeneous processor; implying that:

1. software designers can build hardware; while very little hardware knowledge is required
2. existing software applications can benefit from hardware speedup without the need for an expensive rewrite

The compilers we think are worth mentioning in this category are:

1. ROCCC [31]
2. SPARK [32]
4. Xilinx Vivado [16]
5. LegUp [13]
Important to note is that SPARK seems to be an abandoned project\(^2\) and will not be considered further on. Additionally it is worth mentioning that LegUp is open source for research projects.

We can compare these compilers qualitatively using by several measures:

1. Target domain
2. Target platform
3. Rewriting effort needed to be able to compile to hardware code
4. Application speedup

**Target Domain**

While these products have the same goal, they are targeting different audiences by choosing domains for which they optimise the compilation. The ROCCC compiler optimizes for loop parallelization. The other three compilers do not target specific domains.

In effect ROCCC offers better speedup for kernels in its domain; for applications outside of its domain, the domain specific optimization techniques of these compilers do not improve performance and the compilers are often outperformed by Vivado, LegUp and DWARV [28].

**Target Platform**

There is an additional difference between the compilers in the platforms that they target. Currently, many kinds of platforms are available. One can choose to include FPGA PCI-cards or a processor with an FPGA core besides the CPU core. Additionally, several large vendors are on the market that all sell their own architectures. Xilinx Vivado, for example, only supports their own newer FPGA models (Virtex, Sync and Spartan)\(^3\)

Both ROCCC and DWARV use a platform abstraction layer to be compatible with a wide range of hardware platforms. This is of course a major advantage as it enables customers to use the hardware they have available, thereby lowering the additional costs of utilizing application speedup. LegUp can compile to a hybrid FPGA-based software/hardware system [13].

**Rewriting Effort**

The rewriting effort is an import qualitative measure, because it is a time intensive and expensive process. It is therefore of importance to keep the effort needed to make the software compatible for hardware compilation is kept to the minimal.

The rewriting effort is dependent on the code style and constructs that the compiler supports. ROCCC has very strict requirement and therefore requires a lot of effort to make applications compatible. DWARV, Vivado and LegUp have light requirements and should work well in a short design cycle.

---

\(^2\) The lead developer according to the project page [http://mesl.ucsd.edu/spark/](http://mesl.ucsd.edu/spark/) is no longer working on the project.

Speedup of Applications

As was mentionend in Section 1.4.1, ROCCC optimizes for specific domains. Because of this property, DWARV is outperformed for applications in these domains. Outside these domains Vivado is the best performing compiler, as it consistently performs better than DWARV and LegUp in almost all cases in cycle count, slice usage and execution time [33].

1.4.2. Difficulties in Testing Hardware/Software Co-design

It is generally agreed upon that compiler testing is a difficult subject and cannot be performed perfectly due to the complexity of dynamic semantics\(^4\). In the case of DWARV, testing is further complicated by the fact that DWARV is a software to hardware compiler. Hence:

- Existing compiler test suites are not applicable to DWARV, due to the fact that DWARV does not support all C constructs. This is demonstrated by the fact that the majority of tests of SuperTest, a large compiler testing suite by ACE, failed due to restrictions on input code that compiling for hardware encompasses.
- C-to-HDL compilers are a relatively new research subject. The first industry strength C-to-HDL compiler AutoPilot [34] was released in 2009 by AutoESL (now part of Xilinx), such that few best practices have been developed specifically for these compilers.
- It is more difficult to match VHDL signals to, for example, variables, as there is no set standard on when and where they can be read. Test cases that are to be run on hardware should therefore in general be designed with this in mind.

Due to all this, research related to DWARV always balances on the boundary between computer science and electrical engineering; this field is generally referred to as computer engineering.

1.5. DWARV C-to-VHDL Compiler

DWARV is built on top of the third party compiler development system CoSy\(^5\). CoSy based compilers consist of a set of engines that all manipulate the intermediate representation (IR) of the program[12], which is constructed by the language specific front end engine.

In Section 1.3 we distinguished four compiler stages. In the following sections we will analyze how these stages are implemented in DWARV. An overview can be found in Fig. 1.3.

1.5.1. Lexical and Syntax Analysis and Analysis of Static Semantics

The intermediate representation of the code is generated by the front end engine in a CoSy based compiler. Since DWARV compiles C to HDL, DWARV uses the CFront engine to accomplish this task. CFront is responsible for syntax validation of the code and creating and performing static analysis on the parsing tree, thus implementing the first two stages of the compiler.

As the CFront engine is an ACE product, it is tested by their SuperTest compiler test suite. SuperTest is a test suite with:

\(^4\) The authors of [21] describe how the developed a method for generating randomized test cases for compilers. They used this method to find bugs in major compilers (e.g. GCC) and managed to report 325 previously unknown bugs to the developers.

\(^5\) CoSy is developed by ACE Associated Compiler Experts
Over 3 million tests
Conformance tests
Negative tests
Correctness tests
Compiler quality tests
Regression tests

As CFront is used in many compilers configurations and is tested with such an extensive test suite, we will regard it as sufficiently tested for the scope of this project.

1.5.2. Optimizing Transformations

As in any compiler, DWARV uses a set of optimizing transformations on the IR. These are implemented as separate engines that each implement a single optimization. All optimizing transformations in DWARV are implemented by ACE. The same argument as was made about CFront holds for the optimization engines. Further evaluation of the test quality of these engines is outside the scope of this project and based on the specification of SuperTest, they will be regarded as sufficiently tested.

1.5.3. Code Generation

The code generation (cg) stage is the stage that defines most of the behaviour specific to DWARV. It consists of a set of engines that transform code to deal with the inherent differences between code running on CPU and FPGA cores (such as converting pointers and dealing with the lack of a stack implementation) and engines that lower the representation to VHDL. The engines used for cg are mostly developed by BlueBee. Due to the fact that all custom compiler rengines are in the cg stage, the testing focus is on this stage.

The DWARV test suite contains 96 tests in three basic categories:

**bug tests** (31×) Contains minimal tests that exposes a known bug. After the bug is fixed the test is kept to prevent regression in the future.

**feature tests** (40×) Minimal programs that validate functional requirements of the DWARV compiler. Every functional requirement should have at least one test. If any, edge cases should be covered.
kernel tests (25×) Programs written to test the performance of DWARV and compare performance with other C-to-HDL compilers [12]. These tests simulate real (often computationally intensive) cases.

Additionally the tests can be split into unit and system tests. The bug tests and the largest part of feature tests are set up as unit tests.
2. Goal of the Project

In this chapter, the goal of the project is examined in detail and split into two more specific goals based on a review of the underlying problems.

2.1. Problem Description

In Section 1.3, the need for compiler validation is explained. Although the process of validation is crucial, it is also very complex and time consuming due to the fact that a wide variety of testing methods exist and none of them will yield a suitable level of coverage on its own, as was explained in Section 1.2. Due to this fact, it is importance to perform compiler testing systematically, i.e. perform testing in conformance with a test plan, which describes the methodology that the team will adhere to. This test plan must be designed according to the adequacy criteria that the team has selected in order to maintain a certain test suite adequacy [17, Chapter 20.4].

The existing DWARV test suite, however, was not built based on a complete test plan, nor is there a coverage criterion formulated that the test suite should maintain. These shortcomings have consequences for the quality of DWARV, and consequently the usability and reliability of DWARV, which will be examined in the following section. Additionally, the developers lack adequate tools to collect and maintain the results of running the test suites over time and for different platforms and compiler configurations. This also affects the quality of DWARV, as this makes it difficult to measure and, consequently, improve performance. The exact problems with the available tools and the precise consequences are considered in Section 2.2.2.

2.1.1. Systematic Testing

Although testing with a simple predetermined plan is likely to be incomplete, it is not a problem by definition. The existing test suite could actually be sufficient to validate the correctness of the compiler. It is therefore of importance to be able to have some measure of the quality of the test suite. As no such measure currently exist for the DWARV test suite, the first goal of the project is formulated as follows:

Goal 1. Formulate test suite adequacy criteria and determine, based on these criteria, the quality of the existing DWARV test suite.

Results of the above goal are discussed in Sections 4.1 and 4.2.

Because testing has not been performed systematically, and because code coverage metrics are absent, the developers cannot determine what parts of the compiler code is covered by their tests. It is thus likely that parts of the compiler remain untested or are not tested sufficiently. In order to overcome this we formulated a second goal:

Goal 2. Create a test plan for DWARV based on the on the measured quality of (and the identified issues with) the test suite.
The test plan will serve as an advisory document to the development team.

### 2.1.2. Test Result Management

Given a test suite, a number of different quantifiable metrics can be gathered:

- a measure of the quality of the test suite,
- a measure of the quality of the software under test, and
- a measure of the performance the software under test.

Of the above metrics, our focus is on the first two, because the compiler is not a real-time application and runtime performance of the compiler is therefore not the top priority of BlueBee. Because DWARV is a compiler, there should be a distinction between the runtime performance of DWARV and the performance of the compiled programs. The performance of the compiled programs is directly coupled with the quality of DWARV as with increasing quality, the performance should also increase. To make this distinction clear, we will explicitly distinguish “runtime performance”, which denotes the runtime performance of DWARV, from “performance” or “quality of DWARV” which denotes the performance of the compiled programs.

The second metric is needed to verify that development effort increases the quality of the compiler. The first metric can be considered a measure of the trustworthiness of the second metric.

The quality of the test suite is related to the code coverage (see also Section 1.2.3), as it is a quantifiable means of describing the granularity with which the code has been tested. Since all metrics can, using the right tools, be gathered from data available from running the test suite, the quality and performance of the test suite can be determined from test results. In order to measure performance using a test suite, the tests must be configured to output certain key performance indicators, which we will refer to as **performance metrics**.

Additional to the need to gather the metrics, it is important to be able to create good reports on them in order to draw conclusions about the quality of the software. Ideally, the effort for creating these reports should be as low as possible, such that the developers can make design decisions based on up-to-date quantified measures.

The tools that are currently at the disposal of the team, however, lack several features, which currently makes it time consuming to collect all the necessary metrics and compile helpful reports on them. These shortcomings are examined in Section 2.2.2. A third goal is thus:

**Goal 3.** Improve the test framework for DWARV, such that the development team has access to good reports on the quality of the test suite and the quality of the software under test.

Results of this goal are discussed in Section 4.3.
2.2. Requirements

In Section 2.1, a set of problems were identified that are related to validating DWARV, and in conjunction, goals were formulated that should address these problems. Although all goals serve the same purpose (of improving the quality of the test suite and in effect the quality of DWARV), two distinct goals can be distinguished:

- Creating a test plan.
- Improving the test environment.

In the following sections these goals will be discussed and broken down in requirements.

2.2.1. Creating a Test Plan for DWARV

As has been mentioned, DWARV already has a test suite. Goal 1 was to determine the quality of this test suite, while goal 2 was to propose a test plan for improvement of the suite. These goals can be split into the following requirements:

1. Determine how compiler quality can be validated. This requirement can be split up into the following subtasks:
   - Perform background research on testing compilers.
   - Select methods from this research that are applicable to DWARV and require minimal alterations to the existing test suite.
   - Formulate a quality criterion with which to measure the adequacy of a test suite. Using this criterion, it should be possible to say if a test suite is adequate or not.

2. Determine the current state of the test suite according to the criterion, and where there are shortcomings. This requirement can be divided into two subtasks:
   - Determine the current test suite coverage status.
   - Map the existing tests to the test criterion and measure the adequacy of the test suite.

3. Formulate a test plan with which improve the adequacy of the current test suite and fulfil the quality criterion. This will consist of an advisory report to the CE development group on how to implement the quality assurance of DWARV and can be divided into the following subtasks:
   - Select methods from the background research for improving test suite adequacy and that are applicable to DWARV.
   - Show that the proposed methods are effective in increasing the compiler quality and the test suite adequacy.
2.2.2. Improving the Test Framework

The development team of DWARV maintained a wish list of functional requirements for the test (result) framework. The improvements of the test results framework contribute to goal 3. Furthermore they will aid us in acquiring the data necessary to determine the quality of the current test suite (as formulated in Item 2. Because not every functional requirement contributes to the previously formulated goals, a selection will be made.

The development team expressed the following wishes concerning the test framework:

1. **platforms:** The current infrastructure does not differentiate between target platforms. This implies many sub problems:
   - running a test (suite) for multiple platforms
   - storing results for multiple platforms
   - presenting results for multiple platforms (should probably have 2 type of reports: per platform, per release)

2. **adding metrics:** Have a modular way to add a new metric (for now we have cycle count, frequency estimation, time estimation). Other metrics that we envision to add are: frequency post place and route, time post place and route, area estimation (which can be divided in slice LUT and slice registers). Currently the metrics are spread through multiple scripts/makefiles which makes adding a new one more difficult than it could be.

3. **metric implementation:** Make it easy for the metrics to have a different implementation depending on the platform. (for example on Altera getting the slices would use different tools than for Xilinx)

4. **adding tests:** Make it possible to add a new test result to the stored result of a previous test run. Currently it is necessary to rerun all the tests to create a new result set in case a new test is added to the test suite.

5. **option selection:** The current infrastructure does not have a unified way to treat different options. For example it could “select” the best result from a small set of predefined options.

6. **exploring options:** The current infrastructure does not have an option to explore a large set of options (to determine the best set of parameters for a given test set)

7. **presentation:** There is no easy way to present the information (nicest would be to have a web dashboard, to share results easily between developers/other interested parties).

8. **runtime performance:** Test the performance of the compiler. This could include execution time/memory occupied (this is low priority).

The requirements 1, 2, 3, 4 and 6 contribute to goal 3 and are implemented. Implementing these improvements would significantly decrease the time needed for common tasks performed by the development team and are therefore selected. These improvements and Item 7 were implemented in Dummy, see Section 3.3.
Requirement 5 is omitted because implementing it is not possible within the timespan of the project. We will however add a functional requirement to the framework that will make implementation of it possible in the future.

Requirement 8 “runtime performance” was dropped in accordance with the selection of important metrics in Section 2.1.2.
3. Solution Design and Implementation

The design process and methods used to formulate a solution to the goals set in Chapter 2 will be discussed in this chapter. For the definition of a testing criterion we will first investigate criteria used in science and industry in Section 3.1. Finally, the improvements of the test framework will be discussed in section Section 3.3.

3.1. Adequacy Criteria

CoSy provides the compiler engines for lexical and syntax analysis (stage 1), and the analysis of static semantics (stage 2) as well as some engines in optimizing transformations (stage 3). CoSy engines are tested by ACE, the company that develops CoSy, and are thus not of concern regarding testing DWARV. Of the biggest interest to the development of DWARV is testing for correctness of custom-made transformations in the code generation stage (stage 4).

A sufficient testing solution for DWARV entails certain test adequacy criteria. In the industry there exist testing certifications, such as the Aeronautics DO-178B certificate, which mandates levels of coverage depending on system importance. Among others, DO-178B mandates a 100% statement coverage (see also Section 1.2.3) of most components, and all code that is not covered (and thus cause less than 100% coverage) has to justified. More important code has more extensive coverage requirements, such as complete branch coverage.

For DWARV similar adequacy criteria can be formulated. The different criteria are considered in the following sections. Based on this criteria the adequacy of the test suite can be measured.

3.1.1. Functional Requirement Coverage

Every functional requirement should be covered in the DWARV test suite. Every functional requirement that does not have a test associated with it, has not been validated. This coverage criterion is a refinement of the feature coverage criterion that was already used by the development team.

3.1.2. Compiler Rule and Statement Coverage

The minimal coverage criterion for the test suite considers compiler rule and statement coverage. The criteria can be summarized as follows:

- **rule coverage** Each compiler rule must be used at least once
- **statement coverage** Each statement in the C source files must be executed at least once; every statement that is not executed has to be justified.

These criteria are needed as a fault in a statement (or rule) cannot be revealed without a test that executes it[17, Chapter 12.2].

3.1.3. Language Rule Coverage

The language coverage criteria requires that each supported language rule has been used in the test suite at least once. This is an important criterion to ensure that the compiler contains no trivial bugs for certain language constructs.

It was shown in [35] that minimal test suites for grammar based applications that cover the language rules deliver code coverage comparable to more extensive test suites. They also showed, however, that such a minimal test suite does have reduced fault detection. This is the reason that this adequacy criterion is not enough by itself.

It may seem that the language coverage criterion is only important for testing the first two compiler stages, but this would be a mistake. It is important to determine if the optimizing transformations and code generation can cope with all supported language features.

3.2. Proposed Testing Methodology

In the previous section criteria were formulated for the test suite. In this section we propose a methodology for creating and improving the test suite based on the adequacy criteria. This methodology is designed to uphold the adequacy criteria.

3.2.1. Language Coverage

In order to validate that the compiler creates correct hardware descriptions for all C language constructs, it is important to make sure that the test suite uses all supported language features. The easiest method of ensuring this is to have a minimal test suite that is build specifically for this purpose.

In Section 1.3.1 a method was discussed to generate such a minimal test suite from the language grammar. The C grammar could be used to generate abstract programs using Purdom’s algorithm. The resulting abstract test suite (i.e. test programs in terms of language identifiers) can be used to implement a test suite that adheres to the constraints that DWARV has on the dynamic semantics of the language.

3.2.2. Model-Based Testing

In order to test the transformations of the IR in the code generation stage of the compiler, model based testing is proposed as a test methodology, see for background Section 1.2.2. Model based testing can help the developers to formulate better test cases by limiting the domain to search for meaningful test cases.

Based on the specification of the transformation, a model can be constructed of the kind of input that is touched by the transformation. These models are used to formulate abstract test cases, which in turn can be used to implement actual test cases. The implementation of this testing methodology can be manual or automated. The authors of [36] discuss the process of describing a model in detail in a formal language and automatically generate test cases using the model.

However, implementing the automated approach has high initial costs for the DWARV development team, as both requirements for existing transformations and input models would have to be formalized (many of which are currently not written down in any form). We therefore
propose a less formal approach, based on the requirements and existing implementation of the transformations:

1. Transformations act on specific constructs in the IR, which are matched using matching compiler rules.
2. Based on these matching rules a model can be extracted that models the input on which the transformation acts.
3. The extracted model should be verified using the requirements of the transformation.
4. Using the model, meaningful test (normal, edge and negative edge) cases can be distinguished that cover the possible inputs.

3.2.3. Identification of Missing Test Cases

The above approach of model based testing should yield a high code coverage. This can be verified by running the tests and collecting the code coverage metric afterwards. Using the results, missing test cases can be identified. How this is done in practice, depends on the kind of code that is found to be insufficiently covered. An example of coverage based test case formulation can be found in Section 3.2.

The coverage data is a useful indicator for the quality of the test suite as long as the code coverage is poor (i.e. < 80%), as it requires little effort to employ and is useful for detecting many missing test cases. Even 100% code coverage would not prove the absence of missing test cases though, as blocks of codes could produce incorrect behaviour when combined in different ways than are covered (this is tested instead through integration testing). A more powerful adequacy criterion for the test suite is mutation based adequacy Section 1.2.3.

A more powerful adequacy criterion for the test suite is mutation based adequacy Section 1.2.3. Once the compiler test suite structural code coverage is on suitable level, this method can be employed to further improve the test suite adequacy.

3.2.4. Random Test Case Generation

Although the above two methods combined will provide good means of testing transformations of the IR stage of the compiler, we already argued that it is not sufficient for testing the code generation stage in Section 1.3.4.

The addition of a randomly generated test suite would help to identify hard to find bugs. A program that can create such C-programs is CSmith [21]. It was developed specifically for compiler testing and has been used to find unknown bugs in many mainstream compilers such as gcc. CSmith is very configurable, such that unsupported language features could be turned off. Additionally it is possible to configure the generation probabilities such that it would be possible to stress test DWARV for specific types of programs (e.g. stress test DWARV’s support for structs).
3.3. Testing Framework

The existing test framework used for DWARV development is a combination of a tool called CTest [37], which is part of CMake [38], and a collection of scripts. CMake and CTest are responsible for the organisation of tests into suites, while the custom scripts are used for test execution and result collection and storage. In this scenario, to run a test suite, the DWARV developers invoke CTest with the name of the test suite, and then CTest employs the runner to run the suites. As a final step of the test run, the runner invokes the test result collection scripts. By setting specific environment variables it is possible to control which steps of the testing process are executed. For example, it is possible to disable the collection of performance data, or to prevent test result storage.

Additionally, CTest was able to send the results to CDash, a web application that stores the results (pass/fail) of the tests run. The CDash dashboard provides the developers with a history of tests passing/failing.

Although this system fulfilled the basic requirements for the developers, as DWARV progressed, additional requirements (see Section 2.2.2) were formulated that were difficult to implement. This was due to the following reasons:

1. The format in which the results were written to disk.
2. The lack of separation between runner, result collectors and storage components.
3. The lack of a dedicated tool responsible for invoking the above mentioned components and managing the information flows between them.

It was therefore decided to implement a new tool in Python (version 2.7), which we named Dummy, that would address these shortcomings, and implement the necessary flexibility that was required to be able to manage test results. Indeed, Dummy can be thought of as a **test results framework** rather than just a test framework as it acts as a layer above the existing test framework (see Fig. 3.1).

![Figure 3.1.: Dummy’s place in the testing setup](image)

Note that additional technical documentation is included with Dummy.

3.3.1. Design Overview

It was realized that in general, the testing process can be separated into several stages:

1. **Executing** the tests using the existing test framework/runner.
2. Collection of *metrics* from the execution output per test.
3. Aggregation of statistics over the test metrics.

4. Presentation of the results in the form of collected metrics and statistics.

Tests are often organised in suites that fulfill a certain purpose, such as a regression test suites or a performance test suite.

In a typical development cycle, based on an existing version of some piece of software, a new version is written, and this new version is subsequently committed to a version control system and tested. Hence, it was realized the testing framework should be able to organize test results per version of the software.

In more concrete terms, one could define a certain test suite consisting of programs which should give a certain output. As these programs are run, Dummy should collect the output of the individual tests, by having a “pass/fail”-metric decide whether the output matches the expected output. Note that the expected output is not predetermined by Dummy, but configured by the test developer.

After running all tests, an example statistic could be to calculate the percentage of tests that had a correct output. This percentage can then be presented to the user, and all test results saved to disk.

3.3.2. Per-Component Analysis

We will now look into the components of this testing framework more closely.

Tests and Test Suites

Tests are typically small programs that test an aspect of the behaviour of the software under development. In our particular situation with DWARV, tests consist of pieces of C code which should give an output when compiled with DWARV which is equal to either a theoretically calculated output, or an output as computed when compiled using a more reliable compiler such as GCC.

In principle, it should be possible to at least run the tests against any version of the software, such that it makes sense to compare test results across different versions. This is obstructed, however, by a realistic scenario in which not only the software’s actual source code, but additionally its environment and/or project setup changes over time. We have decided not to deal with this systematically, as it is a fundamental problem resulting from a normal development process.

Tests are organized into test suites by specifying a list of tests and a name. Individual tests are allowed to be part of several test suites; however metrics need only be collected once, and Dummy keeps track of this.
Metrics

Because there are various ways to analyze test runs, we generalized this concept into metrics. For example, one could wish to know whether the output of a test matches with some pre-computed value, which could be decided by a “pass/fail”-metric, but it could also be desirable to know how fast the test ran, which could be collected by a performance-related metric, or how much code of the tested software was hit during execution, which can be collected by a coverage-related metric.

A couple of metrics are included with Dummy, both for demonstration purposes and because we use them in testing DWARV. An example configuration of metrics is given in Section 3.3.3.

Since the metrics are collected by pieces of code, we use the terms “metric”, “metric collector” and “collector” interchangeably.

Statistics

Statistics are usually written specifically for a certain metric. As was already mentioned, it could make sense to compute the percentage of passed tests for the “pass/fail”-metric, but more nuanced data analysis is needed for performance metrics or coverage-related metrics.

Note that whereas metrics are collected for one individual test, statistics are typically collected for a test suite. Statistics can also be thought of as the data reduction step.

Again, a couple of examples are included with Dummy, such as a “coverage” statistic which merges the C coverage data of a test suite, and we give an example configuration in Section 3.3.3.

Test Results

After gathering all data from running the tests and collecting metrics, results are saved on disk for later use. As was mentioned above, in a typical development scenario, usually a specific version of a piece of software is tested, and in Dummy this is supported using thorough integration with Git. Dummy saves test results on disk in the JSON data format sorted by the commit hash of the corresponding version of the software.

Testing fulfills many purposes, depending on the context:

- During development developers will only be interested in pass/fails.
- During test suite adequacy assessment, coverage must be considered.
- During software quality assessment, performance must be considered.

Dummy differentiates between this context by allowing the configuration of different targets. Targets can specify their own values for all dummy configuration options, thus allowing for target-specific metrics. A “coverage” target could for example be configured that collects coverage data. Results are saved in a directory tree under the target name.

Using targets allows the developers to skip coverage collection when running tests during development, which saves time. It also allows the developers of DWARV to run their tests against different hardware platforms and save all the results in a structured manner.
Result Presentation

Because the metrics and statistics are very configurable, there is no general way to further process test results. Hence, Dummy relies on the additional configuration of statistics for smart result presentation, and, simply put, only outputs its internal results storage.

However, since tests are organized in test suites, and test results saved by commit, Dummy allows for a flexible selection of results to be shown. For details on the possible result queries, refer to the Dummy documentation.

Additionally, Dummy separates the aggregation of data to present from the formatting of the data. Per default, Dummy includes several formatters that can be used to format test results or gathered statistics:

- **JsonFormatter**: Outputs the data in the popular JSON format. Useful if the data should be further processed by external scripts
- **LogFormatter**: The default formatter that will print the results in a terminal friendly, human readable format
- **PlotFormatter**: Output (bar/graph) plots of arbitrary numeric metrics
- **LatexFormatter**: Output a latex report on an arbitrary set of test results

It is possible to extend dummy with custom formatters to tailor the precise presentation of the results to your needs.

3.3.3. Basic Dummy Usage

To demonstrate the usage of Dummy in a typical workflow, we will give the steps needed to have Dummy run a test suite, while gathering the coverage data and the pass/fail metric. Additionally we will collect a statistic that counts the number of failed tests. This is also the basic configuration of Dummy for the DWARV project. The more advanced usages of Dummy used for DWARV are demonstrated in the next section.

Dummy is configured using a python module “dummyconfig” that Dummy will look for in the current path and consecutively in the directories above it if it is not found. The fact that it is a python module give the developers the power to define dynamic test suites and utilize the power of a complete programming language. The following example configuration sets up the test suite.

```python
SUITES = {
    # define the 'integration' test suite
    # use globbing to select all tests in the integration directory
    'integration': [
        'integration/*',
        'some/integration/test'
    ]
}
```

Listing 3.1: Example Dummy configuration (Suites)
Furthermore we have to specify what results we would like to gather. In this example we will use collector classes from the Dummy package (which are reference by name), but it also possible to reference a (custom) collector or even use any executable that returns a single value or a JSON encoded object.

MetRICS = {
    'pass/fail': {
        # all predefined collectors (and other configuration classes/functions)
        # can be referenced from the dummy.honeypot module as a shortcut
        'collector': 'dummy.honeypot.PassFailCollector',
    },

    # use the CCoverageCollector to collect coverage data from the source directories
    # you could configure the "PRE_RUN_HOOK" to compile the src with the right
    # CFLAGS before the suite is run
    'coverage': {
        'collector': 'dummy.honeypot.CCoverageCollector',
        'kwargs': {
            # specify where the coverage collector should look for
            # coverage files (.gcno/.gcda in case of C coverage)
            'srcdirs': ['src/'],
        }
    }
}

Listing 3.2: Example Dummy configuration (Metrics)

We can now run the tests from the suite, and store the results by invoking

dummy run --suite integration --store

Listing 3.3: Running a test suite

To distinguish between running the test suite for coverage and running just for pass/fail, we could have put the coverage metric inside the definition of a target, like so:

TARGETS = {
    # create a target cov
    'cov': {
        # merge the default metrics with a new metric
        'METRICS': dict(METRICS,
            coverage=# ... as before
        )
    }
}

Listing 3.4: Example Dummy configuration (Targets)

To collect coverage, we would now have to invoke Dummy like so:

dummy run --suite integration --store --target cov

Listing 3.5: Running Dummy, also invoking coverage

Which would have the results stored in $ROOT/results/<commit_hash>/cov/<testname>/results.json.

Finally the statistics are configured using so-called statistics engines. These are non-specific classes that perform a certain calculation on an arbitrary result collection. Dummy comes with several collectors, such as:
**CountEngine** Calculate how often values occur and calculate a total

**KeyValueEngine** Sum the values per key

**CCoverageEngine** Combine CCoverage outputs

In the next code snippet, a “CountEngine” is used to count the number of tests passing/failing and a “TimerEngine” is used to compute the runtime of the tests:

```python
'passing': {
'engine': 'dummy.honeypot.CountEngine',
'kwargs': {
'metric': 'pass/fail'
}
},
'time': {'engine': 'dummy.honeypot.TimerEngine'}
```

Listing 3.6: Example Dummy configuration (Metrics)

We can now tell dummy to compute the configured statistics for a certain collection of test.

```bash
$ dummy stat --suite integration --target cov
```

# which would output something like this:

```
<table>
<thead>
<tr>
<th>passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIL: 4</td>
</tr>
<tr>
<td>PASS: 1</td>
</tr>
<tr>
<td>total: 5</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration (ended - started): 0:00:03.543352</td>
</tr>
<tr>
<td>duration (test runtime): 0:00:02.107910</td>
</tr>
<tr>
<td>ended: 2013-06-19 11:18:59.016322</td>
</tr>
<tr>
<td>started: 2013-06-19 11:18:55.472970</td>
</tr>
</tbody>
</table>
```

Listing 3.7: Computing a statistic

Additionally we could ask dummy to show us the results per test for a collection of tests.

```bash
$ dummy show --suite integration --target cov --metric pass/fail
```

```
integration/1 (02a6243)
| commit: 02a6243 |
| completed: 2013-06-18 14:01:45.803222 |
| + metrics: |
|   | pass/fail: PASS |
| name: unit_tests/state_unit |
| started: 2013-06-18 14:01:45.743316 |
| target: default |
```

```
integration/2 (02a6243)
| commit: 02a6243 |
| # etc... |
```

Listing 3.8: Showing the pass/fail metric for all tests in the integration suite
3.3.4. Advanced Usage

Although the above example gives an overview of how to use Dummy, it does not show any features that are not in existing test platforms. The power of Dummy is its extensibility and its reconfigurability. The reconfigurability was already partly shown in the previous examples and extends to almost any Dummy feature to provide the flexibility needed to cooperate with existing tools. The extensibility of Dummy applies to:

1. metric collection
2. statistics gathering
3. result presentation

For each of the above components we will discuss an example that was shown to be useful in the collection and analysis of the DWARV test results.

DWARV Metric Extensions

One of the metrics that was collected from the dwarv test output was the amount of clock cycles that were simulated before the result was obtained from the result register. This was done using a shell script in the original setup that used the Unix tool grep to parse the amount of cycles.

In Dummy, metrics are collected using collectors. Collectors can be implemented as a python class to have full access to the python TestResult object, but can also be any executable that calculates the value of a metric. To implement the cycles collector script we could thus simply reuse the original collection script and add a few lines in the Dummy configuration that informed Dummy of the metric and the script to use to collect it.

Additionally, Dummy will pass relevant data to collectors by means of environmental variables (e.g. the test log path, test name, source directory). This ensures that all test specific configuration can be done in one place, thus preventing the test setup from becoming unmaintainable.

Other metrics that were employed in DWARV were:

- **LogCollector** Copies the test execution log to the results directory
- **RulestatCollector** Gathers the number of times that the compiler rules were invoked, for every test

DWARV Statistics Extensions

To analyze the results of the DWARV test suite and to assess the quality of suite, we created a statistics that counts for every compiler rule the frequency of test failures under the condition that that specific rule is used. A value close to 1 would indicate that the rule is only used in tests that fail, while a value close to 0 would mean that the rule is often used in tests that pass.

For large suites with many failing tests, or tests that use many compiler rules (instead of unit testing single rules), this is a useful metric to find rules that contain bugs and prioritize development effort.
Presentation Extensions

Due to the limited time available for the project, no extensions will be written for the presentation of results. However, it is easy to hypothesize some useful and possible extensions:

- A pdf formatter that compiles a human readable summary of the test results
- A CDash formatter that submits a set of results to the CDASH dashboard webservice

Since Dummy result formatters generally extend the Dummy Formatter class, a lot of work is already done for you when you decide to extend Dummy with a custom formatter.
4. Results

The existing test suite of DWARV was analyzed in several ways. First, the existing test methodology was examined qualitatively, of which the results can be found in Section 4.1. In Section 4.2, the results of a quantitative study of the existing test suite can be found. Finally we analyze the results of our attempts to improve the existing test suite based on the methodology proposed in the previous chapter in Section 4.3.

4.1. Qualitative Analysis of the DWARV Test Suite

In this section we will qualitatively discuss the test suite based on the results of an execution of the DWARV test suite. Additionally we will discuss the problems we encountered during execution of the tests.

4.1.1. Test Framework

During execution of the test suite we encountered several problems:

1. Tests would not execute successfully due to errors in the simulation infrastructure that enables simulating generated VHDL.
2. Certain tests exposed bugs that crashed DWARV, which prevented the engine rule coverage from being generated.

To distinguish between the cases were the test execution framework malfunctioned and the case where a test ran successfully but did not return the expected result, we distinguish three test exit statuses:

- **FAIL** The test ran successfully, but did not return the expected result
- **ERROR** The test framework malfunctioned or the test crashed DWARV
- **PASS** The test ran successfully and returned the expected result

This resulted in the result set as listed in Table 4.1. In itself these numbers tell little about the quality of the test suite. However, when noted that 34 out of 38 errors were caused by the malfunctioning VHDL simulation framework, it can be concluded that the execution framework should be improved.
### Table 4.1.: Test results

<table>
<thead>
<tr>
<th>Exit status</th>
<th>Number of Tests</th>
<th>Percentage of Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIL</td>
<td>3</td>
<td>3.1%</td>
</tr>
<tr>
<td>ERROR</td>
<td>34</td>
<td>35.4%</td>
</tr>
<tr>
<td>PASS</td>
<td>59</td>
<td>61.5%</td>
</tr>
</tbody>
</table>

#### 4.1.2. Test Suite Adequacy

Transformations inside the code generation stage in DWARV are currently functionally tested using system tests and unit tests. The methodology specifies that every functional requirement should be validated using a test and that edge cases should also be covered. This is in conformance with the testing criteria that were formulated in Section 3.1.

On inspection of the test suite, it is clear, however, that the existing test suite does not adhere to the language coverage adequacy criterion. Although many C features are used in the test suite, only 4 language features are tested extensively (normal-, edge- and negative cases):

- local arrays
- structs
- constants
- addition of numeric types

And even for these features it is not difficult to find missing cases using the model based testing approach. A test for a nested `struct` appears to be missing for example. Another example is a case that validates the compiler’s behaviour when adding two numeric values of different types. As DWARV supports at least 63% of 38 C language constructs [28], tests are missing for many constructs. Of course, many of these constructs are implicitly tested in other (system) tests, but these are not constructed to test for all relevant cases. In the next section an example construct (double comparison) will be examined qualitatively.

Additionally, the compiler test suite only consists of static, hand-coded tests. The addition of a test suite that tests the compiler with randomly generated C-programs could potentially expose many bugs and would be a valuable addition to the test suite.

The qualitative study helped us to evaluate whether the test suite adheres to adequacy criteria formulated in section 3.1. Based on inspection of the test suite we determined that not every compiler feature (and thus not every functional requirement) is covered with a test. This thus means that criterion 3.1.1 is not fulfilled by the test suite. Additionally we found that many edge cases and common cases of the C language are not covered by the test suite, meaning that criterion 3.1.3 is also not fulfilled.

#### 4.2. Quantitative Analysis of the DWARV Test Suite

The source code of DWARV consists largely of engine functions written in C and special compiler rules, written in a domain specific description language. Compiler rules consist of a name, a header and a body. The headers match specific constructs in the IR and transform them to
VHDL. As part of the analysis of the DWARV test status, we collected both C and rule coverage data.

In this section, this dataset will be summarized and analyzed. Coverage data can be used to:

- determine whether or not a test covers the code that it was intended to test,
- identify missing test cases, and
- identify redundant test cases.

It is important to recognize that although coverage data can be used to identify missing test cases, it cannot prove the completeness of a test suite. Different combinations of executed branches in the code could lead to bugs, even though each branch has been executed by the test suite\(^1\).

We will see how a careful examination of coverage data can lead to an improved test suite in the remainder of this chapter.

### 4.2.1. Rulestat Coverage Statistics

CoSy is shipped with a rulestat engine, which reports the number of times a compiler rule is invoked during the compilation process to the user. We configured Dummy (see Section 3.3) to use this engine to collect the compiler rule coverage data per test and create a statistic that merged these numbers when invoked. By considering this metric, a good measure of the quality of the test suite can be gathered.

Dummy was configured to collect the rule invocation statistics, and compute the total number of times the individual rules are invoked over a complete test suite run. Using this method, a total of 277 rules were identified. An overview of the coverage results can be found in Table A.1.

95 of the rules have a nonzero number of hits, that is, 182 rules are not used in any test. This can be due to a very limited set of types being used in the test suite, and in general indicates a shortcoming of the test suite. We shall review one such shortcoming below, and propose a new set of tests to solve this.

It is important to take care when interpreting the rule statistics. For example, the most frequently invoked rules, usepsr and defpsr, do not emit executable VHDL-code: together, they are used to handle the definition and uses of variables. Hence, rule invocation is not directly correlated with the amount of VHDL code that it emits.

Another example is the rule named if, named after a very fundamental construct in C programs, which is only invoked once. Directly concluding that this implies that the if rule is unsufficienly tested would be a mistake, because the language construct is handled by many different rules, depending on the context. This can be seen if one considers that the related rule if someisnext is invoked 343 times.
Figure 4.1.: Rule usage by the tests that did not have an error before runtime, showing the number of different rules in use.
Rule invocation statistics

A certain C test program invokes various code generation rules. They can also invoke a particular rule multiple times. The number of different rules used is shown in Fig. 4.1, and the total number of rule invocations are shown in Fig. A.1. These charts give a measure of the complexity of various tests: as more compiler rules are used, the unit under test is bigger.

Using this data, we can deduce whether tests can be considered unit tests (although there is no well-defined boundary between unit tests and system tests). For example, the bugs/bug111 test uses 26 different rules, while bugs/bug78 uses only 5. This suggests the former has more diverse code structures. Upon inspecting the code, this seems to be right: whereas bugs/bug78 only does a couple of parallel additions, multiplications and memory dereferences, bugs/bug111 also does left and right bit shifts, and writes back to RAM.

Test failures caused by rule

In this section, we describe how a Dummy statistic can help in inferring which rules frequently cause test failures.

Suppose a new test is added to the test suite, and its output under DWARV is incorrect. Then this could be due to the fact that it tests a rule which does not always emit the right code. In such a situation, we can utilize Dummy to determine which rule likely is at fault, as we will explain.

Many rules are used in the failing test. However, many of them are likely also used in other tests, which executed successfully. Hence, it is unlikely those rules are the cause: they were already tested. So for every rule, we count the number of times it is used in a failing test, and divide that number by the total number of times it is used. More formally, if $E$ is the event that some test fails, and $R$ is the event that some specific code generation rule is used in a test, we compute the probability

$$P(\text{a test fails due to rule } R) = P(E|R) = \frac{P(E \cap R)}{P(R)}$$

To demonstrate the usefulness of this number, let us suppose someone erroneously swapped $rs1$ and $rs2$ in the following code for the div_int_32s (this is a code generation rule that takes care of integer divisions):

```
RULE [div_int_32s] o:mirDiv(rs1:reg, rs2:reg) -> rd:reg;
CONDITION { IS_SINT_A(o.Type) };
EXPAND {
    'lirPscNode psc_init, psc_read;
    psc_init = gcg_create_init_fp_unit(gcg_expand, rs2, rs1, fp_int_div_32s, -1, rd);
    psc_read = gcg_create_read_fp_unit(gcg_expand, rd, fp_int_div_32s, rd);
    'psc_init->ReadPsr = SEQ_INT_copy('gcg_this_psc->ReadPsr);
    'psc_read->ReadPsr = SEQ_INT_copy('gcg_this_psc->ReadPsr);
}
```

Listing 4.1: Rule definition for div_int_32s

\footnote{It is for this reason that a good testing methodology is needed to help the developers identify additional missing test cases.}

40
Any given test for integer division (in our example we wrote `arith/div_int32s`) should now fail. If we now request the `fail_by_rule` statistic, we clearly see which rule are like to have caused test failures:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>div_int32s</code></td>
<td>1.0</td>
</tr>
<tr>
<td><code>init_fp_unit</code></td>
<td>0.0526315789474</td>
</tr>
<tr>
<td><code>read_fp_unit</code></td>
<td>0.047619047619</td>
</tr>
<tr>
<td><code>xreg_write_return</code></td>
<td>0.0232558139535</td>
</tr>
<tr>
<td><code>func_return_next_is_next</code></td>
<td>0.0232558139535</td>
</tr>
</tbody>
</table>

Listing 4.2: Excerpt of `fail_by_rule` statistic

It is possible that a correctly functioning rule is only used in tests that fail, such that $P(E|R) = 1$. In such a case, the `fail_by_rule` statistic will incorrectly indicate it as a likely cause of test failures. However, with a sufficiently large unit test suite, the relation between high values for $P(E|R)$ and malfunctioning rules will be strong, and such a situation is unlikely to occur.

We can conclude that rule utilization statistics are useful to assess what kind of tests are in the suite, whereas the `fail_by_rule` statistic can be used and to assess the origin of test failures.

### 4.2.2. C Coverage

The C coverage data was collected using lcov [39], a front-end for gcov, the GCC tool for code coverage. We built a `CCoverageCollector` that uses lcov to gather coverage results per test case. Additionally, we built a `CCoverageOverviewEngine` to create an overview of the total C coverage of any collection of tests passed to Dummy. The results from the aggregated coverage data is given in Table 4.2.

Even though some engines have excellent coverage, the testing criteria have to be met across all files to be effective. Untested cases within engines are a main cause of concern, because they can easily be executed by a user, who may subsequently run into a bug. A complete overview of the source code coverage can be found in Table A.2. Additionally, an overview of the total amount of lines hit, per test, can be found in Fig. A.2. This image has been generated by Dummy using the `PlotFormatter` for test results.

The branch coverage results are difficult to interpret from a high level overview. Judging from the source code many branches that were not taken correspond to code blocks that were not taken, which suggests that improving the line coverage would in effect also improve the branch coverage. Once the line coverage is on the proper level, branch coverage can be used to further improve the test suite.

### Detailed Analysis

A few top level directories reported low coverage (< 70%):

<table>
<thead>
<tr>
<th>Coverage Type</th>
<th>Lines/Lines Hit</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Coverage</td>
<td>3076/5761</td>
<td>53.5%</td>
</tr>
<tr>
<td>Function Point Coverage</td>
<td>191/262</td>
<td>72.9%</td>
</tr>
<tr>
<td>Branch Coverage</td>
<td>2093/6421</td>
<td>32.5%</td>
</tr>
</tbody>
</table>

Table 4.2.: Code coverage overview of DWARV
hwconfig (50.5%) The hwconfig engine is responsible for reading the configuration file with the target hardware platform configuration. The source for this engine consists of the actual engine description and a grammar file for the configuration language. The former has excellent coverage (97.6%), but the latter causes the overall low coverage percentage (45.8%).

On examination of the grammar file, it is clear that the lacking cases are negative test cases, as the unexecuted code mostly deals with reporting syntax errors.

emit/emit.c (51.6%), fpplib (67.4%), expandec (34.1%), looppragma (18.0%) These source files are written by ACE and extended by DWARV developers for their own purposes. The low coverage is due to the fact that the DWARV test suite does not need to test code that has already been tested by ACE.

Interestingly, one of the things that can be seen from the coverage report on fpplib is that the constructor of the CLibraryManager is used, but the destructor is not. This might indicate the existence of a memory leak.

cgd (43.1%) The cgd directory contains rule based engine definitions. The coverage of engines has already been discussed in Section 4.2.1.

setlatency (49.2%), setperiod (45.7%) Provide timings for the hardware mapping.

4.3. Test Suite Improvements

In Section 3.2 methods were proposed for improving the test suite. In this section we analyse the results by applying the methods to components that were found insufficiently tested.

4.3.1. Speedup of Getting Full Test Suite Results

Because the test results are collected by Dummy for all tests individually, one does not need to rerun the entire suite if code or scripts which only affect one or a couple of tests changes. In effect, this means that the development-testing-evaluation cycle is sped up by use of Dummy.

Together, the individual tests take 7 minutes and 37 seconds to execute. However, this is without Dummy’s processing time, which does a lot of effort to further process coverage data. This is a computationally expensive operation, and Dummy takes 3 hours and 8 minutes to run the entire suite if these are processed.

At first sight, this can be seen as a dramatic deterioration of the runnability of the test suite. However, it takes roughly half a minute to run an individual test with Dummy, and during a typical development cycle one is only interested in the result of one specific test and its code coverage. Hence, if only an individual test needs to be rerun, there is an effective speedup of 15 times to get the same coverage statistics (and more, as we discussed in Section 4.2) in the new test suite and code thanks to the use of Dummy.

Additionally, Dummy can be run with extra command-line options to temporarily disable the collection of certain statistics (such as coverage), such that running individual tests is as fast as without the use of Dummy, and running individual tests takes mere seconds, and the entire test suite runs in a little over 7 minutes and 37 seconds.
4.3.2. Identifying a Missing Test Case Using Rule Coverage

One of the proposed criteria in Section 3.1 was compiler rule coverage. In this example we will show how writing a single test for two compiler rules that were not covered in the existing test suite exposed two separate bugs in DWARV.

The read\textsubscript{data} 64 and write\textsubscript{data} 64 rules were identified as not covered using the rulestat statistics. On inspection of these rules in the source files, it became clear they defined behaviour for memory access of 64 bit variables (such as the long\long data type in C).

Based on this information it was easy to specify two tests (see Appendix B.1) that respectively read or write a long\long from/to memory. To verify that the generated hardware indeed correctly handles the 64 bit variables, the test was set up to test the boundary of a 32 bit long. This was done by creating a long\long variable with the 32 ON bits (value $2^{32} - 1$). Subsequently 1 was added to this number, which should result in the 64-bit value $2^{32}$, or an ON bit in position 33.

By inspection of the rule statistics it was verified that the tests indeed use the rules for which they were written. When compiled with gcc, the result matched the expected $2^{32}$. Interestingly, a simulation of the read test using the VHDL generated by DWARV used to output the wrong value 1. Additionally, the simulation of the VHDL generated for the write\textsubscript{data} 64 test case failed to simulate due to incorrect VHDL.

This was reported and the read test was subsequently repaired in a later version of DWARV. Now the read test passes with the correct ON bit on position 33, but the write test still does not run correctly.

4.3.3. Identifying a Missing Test Case Using the C Coverage

As was earlier identified using the C coverage analysis, the engine hwconfig was insufficiently covered, because negative test cases were missing. An example negative test case (endian\_neg) was created by creating a test case with a custom configuration file that contained a syntax error. Using this test case it was possible to increase the code coverage of hwconfig.

A custom hardware configuration file was created with a faulty setting for endianness, to test the parsing of syntax errors in this case, see Appendix B.2 for the config file. The expected error message was printed, when DWARV was run using this configuration. By checking the coverage it was verified that the line coverage increased from 3076 to 3080 lines executed and that the syntax error related lines were indeed covered.

Since the compiler breaks on the first syntax error, one would need one test per possible syntax error when using system tests to achieve 100% coverage. It is therefore recommended to cover hwconfig using unit-tests that directly invoke the configuration parser.

4.3.4. Model Based Testing

Examining the workings of model based testing will be done on the case of comparisons. By using rulestat coverage it was found that comparisons between certain number types have not been sufficiently tested yet. For example, immediate doubles, 32-bit real numbers and 64-bit
real numbers\(^2\) are not covered, because the rules `cmp_double_imm`, `cmp_real32`, `cmp_real64` have been invoked zero times during the execution of the test suite.

The C language has six relation operators [40]. The operators can be modelled in the following equation:

\[
X \{\text{operator}\} Y \quad | \quad \text{type}(Y) == \text{type}(X)
\]

(4.1)

where \{\text{operator}\} is one of $>$, $\geq$, $<$, $\leq$, $==$ or $!=\$ and the \text{type}(x) function returns to the data type of the variable (e.g. \textbf{int}, \textbf{char}).

The operator can thus only compare two variables of the same type (of course a typecast could be used to cast a variable to the correct type). A relational operator returns the integer 1 for true and 0 if the expression evaluated to false. \(X\) and \(Y\) may be of any C data type.

A few abstract cases that can be derived from the model are listed in table Table 4.3 for the built-in C types. To be able to derive actual test cases from the listed abstract cases, we should also specify what choices can be made when choosing values in test cases that compare valid types. These choices depend on the comparison operator under test. The choices should cover both positive, negative and edge cases. As an example, the choices for the \(<=\) operator can be found in Table 4.4.

If we assume that nine cases per operator is a representative average, a quick calculation shows that for just the valid double comparisons, this would encompass an estimate of 54 test cases!

\[
6 \text{ operators} \cdot 9 \text{ choices} = 54 \text{ cases}
\]

(4.2)

\(^2\)Real here refers to the C floating point types

---

<table>
<thead>
<tr>
<th>Type( (X) )</th>
<th>Operator</th>
<th>Type( (Y) )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>any</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>any</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>any</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>any</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>any</td>
<td>long long</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>any</td>
<td>b</td>
<td>a and b of any built-in type and (a \neq b)</td>
</tr>
</tbody>
</table>

Table 4.3.: Test cases for the comparison.

<table>
<thead>
<tr>
<th>Left Hand Side</th>
<th>Right Hand Side</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>-2</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4.: Possible choices of numeric values for \(<=\) operator.
Many of these cases are currently not in the DWARV test suite. As an example we implemented the test for the double type and \(<=\) operator with the choices in Table 4.4 as the values to be compared. The resulting test can be found in Appendix B.3. The test exposed a bug in DWARV: the code, which is valid C code according to gcc, caused an internal fatal error in DWARV.

The fact that we directly caused the compiler to crash, although we only implemented the test cases for one operator and one data type, suggests that implementing the missing unit test cases for the C language features, as was part of the proposed methodology, is indeed worth the effort.

4.3.5. Random Test Case Generation

The program CSmith [21] was used to test the random test case generation approach described in Section 3.2.4. CSmith is very configurable such that most constructs that are not supported by DWARV can be excluded from the generation process.

To test if CSmith could at all be used to generated code for DWARV we generated a minimal program (∼30 lines; it can be found in Appendix B.4) and tried to compile it with DWARV. This exposed two bugs in DWARV:

- Static function methods are not recognized by DWARV.
- DWARV will exit with a fatal error if the function that should be compiled to hardware takes no arguments (void).

Both bugs occurred in cases that are considered valid, but outside the comfort zone of the compiler. These cases are indeed the kind of cases that randomized tests are intended to expose[21].

Creating Random Test Programs for DWARV

In order to create a proper test from the C code generated by CSmith, several adjustment had to be made to the code:

- The addresses of global variables have to be passed to the functions explicitly.
- No pointers to local variables can be passed to hardware functions.

It is likely that a random program by CSmith would require more adjustments. Since CSmith can, and often will, generate large programs with deep nesting, adjusting them to be good test cases for DWARV could take a lot of time. In a mature compiler it can take many random test cases before a bug is found. The need for manual adjustment of the test cases would thus make random testing a very time consuming task.

Because of time limitation we have not adapted CSmith to automatically generate only supported statements. Future work on random test generation is discussed in Section 5.1.2.

4.4. Discussion

Looking back on the project, we will now discuss the results of the coverage analysis and new test development, respectively in Section 4.4.1 and Section 4.4.2.
4.4.1. Coverage

Efforts were made to make the current coverage of DWARV as complete as possible. However, by the effects of continued development, some tests were unavailable during the period of this project (see also Section 4.1). The effect of this could be an incomplete analysis of the current coverage. It is recommended to make sure that the whole test suite runs properly before writing new tests. This will make sure that failing tests are actually the consequence of bugs in code and not in the tests themselves.

4.4.2. Development of Tests

Due to time constraints the development of tests has been limited to a few examples (see Section 4.3). Originally we would have like to developed a more significant amount of tests, based on our own testing methodology. This way the effectiveness of our method would have been proven on a larger set of tests. Our test development examples show how coverage can be structurally improved using the delivered tools.
5. Conclusion

Testing DWARV is a complicated exercise due to interaction between hardware and software. After performing research into the many required subjects we have defined a set of goals for the project:

1. Rate the adequacy of the existing DWARV test suite based on formulated adequacy criteria.
2. Create a test plan based on identified shortcomings of the test suite.
3. Improve the test framework.

Here, the first two items are the requirements of Section 2.2.1, and the last item summarizes the requirements of Section 2.2.2.

Based on extensive background research, several key adequacy criteria were formulated:

1. Supported C language coverage
2. Compiler rule coverage
3. Structural code coverage
4. Mutation detection adequacy

This fulfills goal 1.

Furthermore, we formulated several methods for extending the test suite such that the adequacy criteria can be fulfilled and suggested a methodology for validating the above adequacy criteria. We suggested the use of

1. Purdom’s algorithm on a grammar of the supported C subset, and
2. model based testing for systematically testing compiler transformations, and
3. coverage (and mutation) analysis to find missing test cases, and
4. random testing for finding difficult to detect compiler bugs.

The above completes goal 2.

For the third and final goal of Chapter 2, we implemented a tool called Dummy to enhance the test framework, by creating a means of managing and analysing test results. We’ve applied the use of Dummy to DWARV to obtain our measures of the test suite quality. We also implemented several useful statistics and showed that they can be used to streamline the debugging process of DWARV. Dummy has been built with extensibility as a prime feature, such that it can be tailored to the future needs of the DWARV team, and more generally, other compiler development teams.
5.1. Future Work

In this section, several possible foci of future research are examined.

5.1.1. Correctness of Compiler Transformations

To the best of our knowledge a practical method for showing correctness of compiler transformations is still unavailable. [26, 27] show an impractical theoretical method (see also Section 1.3.3), while [21] shows a more practical but non-structural randomised method. Further research will have to find a practical and structural method for sufficiently showing the correctness of compiler transformations.

5.1.2. Random Testing

Random testing was performed using CSmith and manual adjustments. Because some C structures are not support by DWARV, the random testing environment has to be configured to keep to these limitations. Three solutions can be thought of:

1. If possible, configure CSmith not to generate the unsupported language features. This is supported in CSmith for a number of language features.
2. Use the C-to-C compilation that is already part of the DWARV test framework to make the necessary adjustments.
3. Adjust the CSmith code generation to output correct test cases\(^1\).

5.1.3. Employing Fault Based Testing

In Dummy and the associated metric collectors and statistics engines, we have provided the tools for the development team to employ structural testing. We believe this method will prove very useful for improving upon the current state of the test suite, as we identified many missing tests. In the future however, it might be necessary to use more stringent adequacy analysis, such as the recommended mutation analysis. How to apply this to DWARV in a suitable manner will require further research.

---

\(^1\) CSmith is open source.
Bibliography


49


R. Nane, “[a comparison between dwarv 3.0, dwarv 2.0, legup 2.0 and vivado hls],” the title is not chosen yet.


# A. Coverage Report

## A.1. Rulestat Coverage

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Count</th>
<th>Instruction</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefPsr_reg</td>
<td>0</td>
<td>MultiDef_reg</td>
<td>523</td>
</tr>
<tr>
<td>NoRule</td>
<td>0</td>
<td>UsePsr_reg</td>
<td>0</td>
</tr>
<tr>
<td>add_double_double</td>
<td>0</td>
<td>add_double_imm</td>
<td>0</td>
</tr>
<tr>
<td>add_double_immX</td>
<td>0</td>
<td>add_fixed</td>
<td>0</td>
</tr>
<tr>
<td>add_int_s</td>
<td>376</td>
<td>add_int_simm</td>
<td>353</td>
</tr>
<tr>
<td>add_int_simmX</td>
<td>0</td>
<td>add_int_u</td>
<td>61</td>
</tr>
<tr>
<td>add_int_uimm</td>
<td>21</td>
<td>add_int_uimmX</td>
<td>0</td>
</tr>
<tr>
<td>add_real_imm</td>
<td>0</td>
<td>add_real_immX</td>
<td>0</td>
</tr>
<tr>
<td>add_real_real</td>
<td>147</td>
<td>addr_global</td>
<td>0</td>
</tr>
<tr>
<td>addrdiff</td>
<td>0</td>
<td>addrplus</td>
<td>2047</td>
</tr>
<tr>
<td>addrplus_imm</td>
<td>1222</td>
<td>addrplus_immX</td>
<td>0</td>
</tr>
<tr>
<td>aggr_assign</td>
<td>0</td>
<td>and_bool</td>
<td>12</td>
</tr>
<tr>
<td>and_int_s</td>
<td>3</td>
<td>and_int_u</td>
<td>290</td>
</tr>
<tr>
<td>and_simm</td>
<td>2</td>
<td>and_simmX</td>
<td>0</td>
</tr>
<tr>
<td>and_nimm</td>
<td>0</td>
<td>and_nimmX</td>
<td>0</td>
</tr>
<tr>
<td>boolval</td>
<td>14</td>
<td>boolval</td>
<td>10</td>
</tr>
<tr>
<td>boolval_reg</td>
<td>14</td>
<td>branchpred</td>
<td>0</td>
</tr>
<tr>
<td>cmp_cc</td>
<td>80</td>
<td>cmp_double_imm</td>
<td>0</td>
</tr>
<tr>
<td>cmp_real32</td>
<td>0</td>
<td>cmp_real32_pred</td>
<td>2</td>
</tr>
<tr>
<td>cmp_real64</td>
<td>0</td>
<td>cmp_real_imm</td>
<td>0</td>
</tr>
<tr>
<td>cmp_reg</td>
<td>4</td>
<td>cmpi_cc</td>
<td>264</td>
</tr>
<tr>
<td>cvn_double_reg</td>
<td>20</td>
<td>cvn_double_real</td>
<td>0</td>
</tr>
<tr>
<td>cvn_double_schar</td>
<td>0</td>
<td>cvn_double_sint</td>
<td>0</td>
</tr>
<tr>
<td>cvn_double_sll</td>
<td>0</td>
<td>cvn_double_sshort</td>
<td>0</td>
</tr>
<tr>
<td>cvn_double_uchar</td>
<td>0</td>
<td>cvn_double_uint</td>
<td>0</td>
</tr>
<tr>
<td>cvn_double ull</td>
<td>0</td>
<td>cvn_double_ushort</td>
<td>0</td>
</tr>
<tr>
<td>cvn_real_double</td>
<td>0</td>
<td>cvn_real_schar</td>
<td>0</td>
</tr>
<tr>
<td>cvn_real_sint</td>
<td>12</td>
<td>cvn_real_sll</td>
<td>0</td>
</tr>
<tr>
<td>cvn_real_short</td>
<td>0</td>
<td>cvn_real_uchar</td>
<td>0</td>
</tr>
<tr>
<td>cvn_real_uint</td>
<td>0</td>
<td>cvn_real_all</td>
<td>0</td>
</tr>
<tr>
<td>cvn_real_ushort</td>
<td>0</td>
<td>cvn_schar_double</td>
<td>0</td>
</tr>
<tr>
<td>cvn_schar_real</td>
<td>4</td>
<td>cvn_sint</td>
<td>0</td>
</tr>
<tr>
<td>cvn_sint_double</td>
<td>0</td>
<td>cvn_sint_ptr</td>
<td>327</td>
</tr>
<tr>
<td>cvn_sint_real</td>
<td>24</td>
<td>cvn_sll_double</td>
<td>0</td>
</tr>
<tr>
<td>cvn_sll_real</td>
<td>0</td>
<td>cvn_sshort_double</td>
<td>0</td>
</tr>
<tr>
<td>cvn_sshort_real</td>
<td>4</td>
<td>cvn_uchar_double</td>
<td>0</td>
</tr>
<tr>
<td>cvn_uchar_real</td>
<td>2</td>
<td>cvn_uint</td>
<td>0</td>
</tr>
<tr>
<td>cvn_uint_double</td>
<td>0</td>
<td>cvn_uint_ptr</td>
<td>122</td>
</tr>
<tr>
<td>cvn_uint_real</td>
<td>6</td>
<td>cvn ull_double</td>
<td>0</td>
</tr>
<tr>
<td>cvn ull_real</td>
<td>0</td>
<td>cvn_ushort_double</td>
<td>0</td>
</tr>
<tr>
<td>cvn_ushort_real</td>
<td>0</td>
<td>convert_fixed2float</td>
<td>0</td>
</tr>
<tr>
<td>Instruction</td>
<td>Opcode</td>
<td>Count</td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>convert_fixed2int</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>convert_int2fixed</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data_addr_64</td>
<td>0</td>
<td>1134</td>
<td></td>
</tr>
<tr>
<td>defpsr</td>
<td>3540</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_double</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_32s</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_8s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_double</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_int_16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_int_32u</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_imm_int_8u</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_int_16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_int_32s</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_int_64</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_int_8u</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>div_real_imm</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_simm_p2</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_uimm_p2</td>
<td>0</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>evaluate</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>func_return_next_is_next</td>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gce_jump</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td>289</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_ins</td>
<td>0</td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>init_fp.imm_unit</td>
<td>0</td>
<td>490</td>
<td></td>
</tr>
<tr>
<td>init.fpconv_unit</td>
<td>59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init.fplogic_unit_pred</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init.imm_fp_unit</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init.imm_unit</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init.proc.param_unit</td>
<td>0</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>ld_32</td>
<td>1134</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld_fixed_le32bit</td>
<td>0</td>
<td>731</td>
<td></td>
</tr>
<tr>
<td>ld_localmem2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldi_reg_fixed</td>
<td>0</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>ldi_reg_rim64</td>
<td>0</td>
<td>1322</td>
<td></td>
</tr>
<tr>
<td>ldi_reg_simm</td>
<td>687</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mirsel_then</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mirsel_thenelse</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_imm_16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_imm_32s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_imm_8s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_imm_int_16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_imm_int_32s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_imm_int_8s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_int_16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_int_32s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_int_8s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_simm_p2</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov_reg</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpy_double_double</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpy_double_imm</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpy_int_168s</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpy_int_32s</td>
<td>78</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Count</td>
<td>Function</td>
<td>Count</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>mpy_int_64s</td>
<td>0</td>
<td>mpy_int_64u</td>
<td>2</td>
</tr>
<tr>
<td>mpy_real_imm</td>
<td>0</td>
<td>mpy_real_immX</td>
<td>0</td>
</tr>
<tr>
<td>mpy_real_real</td>
<td>185</td>
<td>mpy_simm</td>
<td>572</td>
</tr>
<tr>
<td>mpy_simmX</td>
<td>0</td>
<td>mpy_simm_64s</td>
<td>2</td>
</tr>
<tr>
<td>mpy_simm_64sX</td>
<td>0</td>
<td>mpy_uimm</td>
<td>0</td>
</tr>
<tr>
<td>mpy_uimmX</td>
<td>0</td>
<td>mpy_uimm_64u</td>
<td>0</td>
</tr>
<tr>
<td>mpy_uimm_64uX</td>
<td>0</td>
<td>mul.fixed</td>
<td>0</td>
</tr>
<tr>
<td>neg_fp_32</td>
<td>7</td>
<td>neg_fp_64</td>
<td>0</td>
</tr>
<tr>
<td>neg_int</td>
<td>29</td>
<td>nogoto</td>
<td>359</td>
</tr>
<tr>
<td>not_double</td>
<td>0</td>
<td>not_int</td>
<td>105</td>
</tr>
<tr>
<td>not_real</td>
<td>0</td>
<td>or_bool</td>
<td>0</td>
</tr>
<tr>
<td>or_int_s</td>
<td>4</td>
<td>or_int_u</td>
<td>634</td>
</tr>
<tr>
<td>or_simm</td>
<td>0</td>
<td>or_simmX</td>
<td>0</td>
</tr>
<tr>
<td>or_uimm</td>
<td>0</td>
<td>or_uimmX</td>
<td>0</td>
</tr>
<tr>
<td>proc_call_direct</td>
<td>0</td>
<td>prologue</td>
<td>158</td>
</tr>
<tr>
<td>read_data</td>
<td>0</td>
<td>read_data_64</td>
<td>0</td>
</tr>
<tr>
<td>read_data_mbram</td>
<td>1134</td>
<td>read_fp_unit</td>
<td>552</td>
</tr>
<tr>
<td>read_fpconv_unit</td>
<td>52</td>
<td>read_fplogic_unit</td>
<td>0</td>
</tr>
<tr>
<td>read_fplogic_unit_pred</td>
<td>2</td>
<td>read_func_res_unit</td>
<td>0</td>
</tr>
<tr>
<td>ref_to_array</td>
<td>1535</td>
<td>reg_to_register_arg</td>
<td>0</td>
</tr>
<tr>
<td>rev_cc</td>
<td>0</td>
<td>rimm</td>
<td>0</td>
</tr>
<tr>
<td>set_stackpointer</td>
<td>0</td>
<td>shl_imm</td>
<td>514</td>
</tr>
<tr>
<td>shl_imm_int</td>
<td>12</td>
<td>shl_int</td>
<td>18</td>
</tr>
<tr>
<td>simm</td>
<td>1565</td>
<td>sra_imm</td>
<td>0</td>
</tr>
<tr>
<td>sra_imm_int</td>
<td>0</td>
<td>sra_int</td>
<td>67</td>
</tr>
<tr>
<td>srl_imm</td>
<td>0</td>
<td>srl_imm_int</td>
<td>0</td>
</tr>
<tr>
<td>srl_int</td>
<td>376</td>
<td>st_localmem</td>
<td>1702</td>
</tr>
<tr>
<td>stack_allocation</td>
<td>0</td>
<td>streg</td>
<td>512</td>
</tr>
<tr>
<td>streg_fp32_imm</td>
<td>0</td>
<td>streg_fp64_imm</td>
<td>0</td>
</tr>
<tr>
<td>streg_imm</td>
<td>6</td>
<td>sub_double</td>
<td>0</td>
</tr>
<tr>
<td>sub_double_imm</td>
<td>0</td>
<td>sub_imm_double</td>
<td>0</td>
</tr>
<tr>
<td>sub_imm_real</td>
<td>0</td>
<td>sub_int_simm</td>
<td>44</td>
</tr>
<tr>
<td>sub_int_uimm</td>
<td>0</td>
<td>sub_real</td>
<td>53</td>
</tr>
<tr>
<td>sub_real_imm</td>
<td>0</td>
<td>sub_simm_int</td>
<td>36</td>
</tr>
<tr>
<td>sub_sint</td>
<td>147</td>
<td>sub_uimm_int</td>
<td>0</td>
</tr>
<tr>
<td>sub_uint</td>
<td>11</td>
<td>subscript_assign</td>
<td>0</td>
</tr>
<tr>
<td>subscript_assign_imm</td>
<td>0</td>
<td>subscript_assign_mbram</td>
<td>0</td>
</tr>
<tr>
<td>subscript_assign_mbram_imm</td>
<td>0</td>
<td>subscript_read</td>
<td>0</td>
</tr>
<tr>
<td>subscript_read_addr</td>
<td>0</td>
<td>subscript_read_addr_imm</td>
<td>0</td>
</tr>
<tr>
<td>subscript_read_imm</td>
<td>0</td>
<td>subscript_read_mbram</td>
<td>0</td>
</tr>
<tr>
<td>subscript_read_mbram_imm</td>
<td>0</td>
<td>total</td>
<td>40666</td>
</tr>
<tr>
<td>uimm</td>
<td>0</td>
<td>usepsr</td>
<td>11276</td>
</tr>
<tr>
<td>vastart</td>
<td>0</td>
<td>write_data_64</td>
<td>0</td>
</tr>
<tr>
<td>write_data_imm64</td>
<td>0</td>
<td>xor_int_s</td>
<td>4</td>
</tr>
<tr>
<td>xor_int_u</td>
<td>1716</td>
<td>xor_simm</td>
<td>0</td>
</tr>
<tr>
<td>xor_simmX</td>
<td>0</td>
<td>xor_uimm</td>
<td>0</td>
</tr>
<tr>
<td>xor_uimmX</td>
<td>0</td>
<td>xreg_write_return</td>
<td>85</td>
</tr>
</tbody>
</table>

Table A.1.: Coverage of the code generation rules
Figure A.1.: Rule usage by the tests that did not have an error before runtime, showing the total number of times a rule is invoked. Some of the bars of the rule invocations stretch far beyond this plot; e.g. the kernels/idct test uses 6124 rule invocations.
A.2. Code Coverage

Table A.2.: A more detailed overview of the coverage data on DWARV.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Line Coverage</th>
<th>Function Point Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg/cgd</td>
<td>43.1%</td>
<td>50.0%</td>
<td>34.6%</td>
</tr>
<tr>
<td>cg/checkpragma/src</td>
<td>96.5%</td>
<td>100.0%</td>
<td>79.6%</td>
</tr>
<tr>
<td>cg/emit/src</td>
<td>51.6%</td>
<td>85.2%</td>
<td>12.2%</td>
</tr>
<tr>
<td>cg/fplib/src</td>
<td>67.4%</td>
<td>69.2%</td>
<td>47.6%</td>
</tr>
<tr>
<td>cg/hwconfig/src</td>
<td>50.5%</td>
<td>88.2%</td>
<td>29.5%</td>
</tr>
<tr>
<td>cg/localrequiv/src</td>
<td>88.9%</td>
<td>100.0%</td>
<td>59.5%</td>
</tr>
<tr>
<td>cg/pseudorequiv/src</td>
<td>87.0%</td>
<td>91.7%</td>
<td>73.7%</td>
</tr>
<tr>
<td>cg/rex/src</td>
<td>86.2%</td>
<td>100.0%</td>
<td>67.9%</td>
</tr>
<tr>
<td>cg/setlatency/src</td>
<td>49.2%</td>
<td>80.0%</td>
<td>39.6%</td>
</tr>
<tr>
<td>cg/setlocarr/src</td>
<td>97.9%</td>
<td>100.0%</td>
<td>80.6%</td>
</tr>
<tr>
<td>cg/setmbramtypes/src</td>
<td>93.9%</td>
<td>100.0%</td>
<td>65.6%</td>
</tr>
<tr>
<td>cg/setperiod/src</td>
<td>45.7%</td>
<td>75.0%</td>
<td>19.2%</td>
</tr>
<tr>
<td>cg/setregisters/src</td>
<td>16.7%</td>
<td>33.3%</td>
<td>3.5%</td>
</tr>
<tr>
<td>cg/sizecalc/src</td>
<td>89.9%</td>
<td>100.0%</td>
<td>85.2%</td>
</tr>
<tr>
<td>expandec/src</td>
<td>34.1%</td>
<td>66.7%</td>
<td>40.0%</td>
</tr>
<tr>
<td>looppragma/src</td>
<td>18.0%</td>
<td>38.5%</td>
<td>8.8%</td>
</tr>
<tr>
<td>pointerconvert/src</td>
<td>83.6%</td>
<td>100.0%</td>
<td>58.1%</td>
</tr>
</tbody>
</table>

A.2.1. C Line Coverage Per Test

56
Figure A.2.: Line coverage per test
B. Test Cases

B.1. Reading/Writing 64 Bit Blocks

```c
#ifdef dwarv
  void* malloc( int );
#else
#include <stdlib.h>
#include <stdio.h>
#endif

#pragma bb map
void sum( long long *a, long long *b, long long *rez ) {
  *rez = *a + *b;
}

int main( void ) {
  long long *in_a, *in_b, *rez;
  in_a = (long long*) malloc( 8 );
  in_b = (long long*) malloc( 8 );
  rez = (long long*) malloc( 8 );
  *in_a = 1L;
  *in_a = (*in_a)<<32;
  *in_b = 1;

  sum( in_a, in_b, rez );
#ifdef dwarv
  printf( "%lld", *rez );
#endif
  return 0;
}
```

Listing B.1: Test application for 64 bit memory reads.

```c
void* malloc( int );

#pragma bb map
void sum( long long a, long long b, long long* rez ) {
  *rez = a + b;
}

int main( void ) {
  long long in_a, in_b;
  long long* rez = (long long*) malloc( sizeof( long long ));
  in_a = 0xFFFFFFFF;
  in_b = 1;

  sum( in_a, in_b, rez );
  return 0;
}
```

Listing B.2: Test application for 64 bit memory writes.
B.2. Hwconfig Negative Test

```
endian = nonsenseb
** in bits
byte_size = hoi8
** hex value
xreg_start_address_in = doi0
** in bits
xreg_address_size = doc10
** in bytes
xreg_word_size = doep4
xreg_read_cycles = b4
xreg_rw_burst = bon
memory_read_cycles = ij4
memory_rw_burst = ijon
** in bits
memory_address_size = ij18
** in bytes
memory_word_size = ij4
** in bytes
char = iij1
** in bytes
signed_char = iij1
** in bytes
unsigned_char = iij1
** in bytes
short = iij2
** in bytes
unsigned_short = iij2
** in bytes
int = iij4
** in bytes
unsigned_int = iij4
** in bytes
long = iij4
** in bytes
unsigned_long = iij4
** in bytes
long_long = iij8
** in bytes
unsigned_long_long = iij8
```

Listing B.3: A configuration file with a syntax error for endianness.

B.3. Model Based Testing

```
#include <stdio.h>

#pragma bb map
int comp( double left, double right ) {
    return left <= right;
}

int main( void ) {
    int rez;
    rez =  comp( 1.0, 2.0 ) &&
        ! comp( 2.0, 1.0 ) &&
        comp( 0.0, 0.0 ) &&
        comp( 0.0, 1.0 ) &&
        comp( -1.0, 0.0 ) &&
        comp( -1.0, 2.0 ) &&
        ! comp( 1.0, -2.0 ) &&
        comp( -2.0, -1.0 ) &&
        ! comp( -1.0, -2.0 );
```
Listing B.4: Model based test for immediate double comparisons.

B.4. Random Test Case Generation

/*
 * This is a RANDOMLY GENERATED PROGRAM.
 * Generator: csmith 2.1.0
 * Git version: exported
 * Options: --no-pointers --no-unions
 * Seed: 1622383534
 */
#include "csmith.h"

static long _undefined;

#pragma map generate_hw
int32_t func1 (void)
{
    int64_t l2 = 1L;
    return l2;
}

int main (int argc, char* argv[])
{
    int print_hash_value = 0;
    if (argc == 2 && strcmp(argv[1], "1") == 0) print_hash_value = 1;
    platform_main_begin();
    crc32_gentab();
    #pragma map call_hw VIRTEX5_0
    func1();
    platform_main_end(crc32_context * 0xFFFFFFFFUL, print_hash_value);
    return 0;
}

/******************** statistics ********************
Listing B.5: Randomly generated test case. (Trimmed statistics)