Integration of existing optimisation techniques with the DWARV C-to-VHDL compiler

Marcel Slotema

Abstract

Hardware acceleration using reconfigurable devices is a hot research item. To facilitate this acceleration technique, also called reconfigurable computing, tools are being developed that translate High-Level Languages into Hardware Description Languages. DWARV is such a tool developed at Delft University of Technology, translating C into VHDL.

The way C code is written affects the performance of DWARV significantly. Manual modifications can improve the speedup of that kernel significantly. This thesis will work towards closing the performance gap between modified sources and unmodified sources. By integrating different optimisation techniques, less manual modifications will need to be made for improved performance. The techniques covered by this thesis are loop unrolling, loop invariant code motion, software-sided caching and algebraic simplifications.

Integrating these techniques into DWARV was done, taking into account possible dependencies between the different optimisation techniques. When combining all these techniques together, speedups of 2.8 were achieved for individual kernels. On average, a kernel is 1.45 times faster when the optimisation techniques from this thesis are used.
Integration of existing optimisation techniques with the DWARV C-to-VHDL compiler

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Marcel Slotema
born in Rotterdam, The Netherlands
Integration of existing optimisation techniques with the DWARV C-to-VHDL compiler

by Marcel Slotema

Abstract

Hardware acceleration using reconfigurable devices is a hot research item. To facilitate this acceleration technique, also called reconfigurable computing, tools are being developed that translate High-Level Languages into Hardware Description Languages. DWARV is such a tool developed at Delft University of Technology, translating C into VHDL.

The way C code is written affects the performance of DWARV significantly. Manual modifications can improve the speedup of that kernel significantly. This thesis will work towards closing the performance gap between modified sources and unmodified sources. By integrating different optimisation techniques, less manual modifications will need to be made for improved performance. The techniques covered by this thesis are loop unrolling, loop invariant code motion, software-sided caching and algebraic simplifications.

Integrating these techniques into DWARV was done, taking into account possible dependencies between the different optimisation techniques. When combining all these techniques together, speedups of 2.8 were achieved for individual kernels. On average, a kernel is 1.45 times faster when the optimisation techniques from this thesis are used.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-13

Committee Members :

Advisor: Koen Bertels, CE, TU Delft
Chairperson: Koen Bertels, CE, TU Delft
Member: Zaid Al-Ars, CE, TU Delft
Member: Johan Pouwelse, PDS, TU Delft
Member: Arnaldo Azevedo, CE, TU Delft
# Contents

**List of Figures**  
vi

**List of Tables**  
ix

**Acknowledgements**  
xi

1 Introduction  
1.1 Problem description  
1.2 Thesis goals  
1.3 Thesis organisation  
2 Related Work  
2.1 Introduction  
2.2 Hardware Compilers  
2.2.1 Criteria used for evaluation  
2.2.2 GCC2Verilog  
2.2.3 High Level Synthesis using the Intermediate Predicate Format  
2.2.4 LegUp  
2.2.5 The VEX VLIW compiler in combination with Handel-C  
2.2.6 Quick summary  
2.3 Compiler Optimisation Space Exploration  
2.3.1 The Compiler Optimization Level Exploration framework  
2.3.2 Optimizing General Purpose Compiler Optimizations  
2.3.3 Machine Learning Compiler  
2.3.4 Exhaustive Optimization Phase Order Space Exploration  
2.3.5 Compiler Optimization-Space Exploration  
2.3.6 Quick summary  
2.4 Conclusion  
3 Selection of CoSy Compiler Optimisations  
3.1 Introduction  
3.2 Criteria to evaluate generated VHDL  
3.2.1 Execution Time  
3.2.2 Used area  
3.2.3 Power considerations  
3.3 Requirements for optimisations  
3.4 Structure of DWARY  
3.5 Available CoSy engines  
3.6 Selection of optimisation engines  
3.7 Conclusion
4 Preliminary results 39
  4.1 Introduction ...................................................... 39
  4.2 Kernels used for evaluation ..................................... 39
  4.3 Results for an initial placement of the selected engines ........ 43
    4.3.1 Experimental setup ......................................... 43
    4.3.2 Results ...................................................... 44
  4.4 Dependencies between engines ..................................... 46
    4.4.1 Description of dependencies ................................ 46
    4.4.2 Results ...................................................... 50
  4.5 Conclusion .......................................................... 51

5 Integration of selected optimisation engines with DWARV 53
  5.1 Introduction .......................................................... 53
  5.2 Loop unrolling ....................................................... 53
    5.2.1 Introduction .................................................. 53
    5.2.2 Options ......................................................... 54
    5.2.3 Implementation ................................................ 57
  5.3 Loop invariant code motion ........................................ 62
    5.3.1 Introduction .................................................. 62
    5.3.2 Options ......................................................... 63
    5.3.3 Implementation ................................................ 66
  5.4 Software-sided caching ............................................. 67
    5.4.1 Introduction .................................................. 67
    5.4.2 Options ......................................................... 68
    5.4.3 Implementation ................................................ 70
  5.5 Algebraic simplifications ........................................... 71
    5.5.1 Introduction .................................................. 71
    5.5.2 Options ......................................................... 71
    5.5.3 Implementation ................................................ 72
  5.6 Conclusion .......................................................... 73

6 Results .................................................................. 75
  6.1 Introduction .......................................................... 75
  6.2 Loop-unrolling ......................................................... 75
    6.2.1 Introduction .................................................... 75
    6.2.2 Evaluation of improvements .................................... 75
    6.2.3 Exploring the maxfactor option ............................... 78
  6.3 Loop invariant code motion ......................................... 81
    6.3.1 Introduction .................................................... 81
    6.3.2 Evaluation of improvements .................................... 81
    6.3.3 Exploring the limit option ..................................... 82
    6.3.4 Comparing the loophoist engine against the loopinvariant engine 83
  6.4 Software-sided caching ............................................. 84
    6.4.1 Introduction .................................................... 84
    6.4.2 Evaluation of improvements .................................... 84
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Results of the initial placement of the non-loop-optimising engines</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Results of the initial placement of the loop-optimising engines</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Results of the placement of the non-loop-optimising engines, taking into</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>account dependencies</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Results of the placement of the loop-optimising engines, taking into account</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>dependencies</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of dependencies concerning the <code>loopunroll</code> engine</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of dependencies concerning the <code>loopunhoist</code> and <code>loopinvariant</code></td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Overview of dependencies concerning the <code>loopscalar</code> and <code>cache</code> engines</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>Overview of dependencies concerning the <code>algebraic</code> engine</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of DWARV without (baseline) and with loop-unrolling</td>
<td>76</td>
</tr>
<tr>
<td>6.2</td>
<td>Influence of the <code>maxfactor</code> option on the execution time</td>
<td>78</td>
</tr>
<tr>
<td>6.3</td>
<td>Influence of the <code>maxfactor</code> option as powers of two on the execution time</td>
<td>79</td>
</tr>
<tr>
<td>6.4</td>
<td>The average speedup as a function of the maximum unrolling factor</td>
<td>80</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison of the loop invariant code motion engines</td>
<td>81</td>
</tr>
<tr>
<td>6.6</td>
<td>Influence of the <code>limit</code> option on the speedup</td>
<td>83</td>
</tr>
<tr>
<td>6.7</td>
<td>Comparison of the <code>loophoist</code> engine with the <code>loopinvariant</code> engine</td>
<td>84</td>
</tr>
<tr>
<td>6.8</td>
<td>Comparison of the software caching engines</td>
<td>85</td>
</tr>
<tr>
<td>6.9</td>
<td>Comparison of the <code>algebraic</code> engine</td>
<td>87</td>
</tr>
<tr>
<td>6.10</td>
<td>Average speedup of different orderings of four of the optimisation engines</td>
<td>88</td>
</tr>
<tr>
<td>6.11</td>
<td>Detailed speedup of different orderings of four of the optimisation engines</td>
<td>90</td>
</tr>
<tr>
<td>6.12</td>
<td>Comparison of DWARV with LegUp</td>
<td>91</td>
</tr>
<tr>
<td>6.13</td>
<td>Impact of optimisations for DWARV and LegUp</td>
<td>92</td>
</tr>
<tr>
<td>7.1</td>
<td>Average speedup of the different optimisation engines</td>
<td>95</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Quick summary of the different tools</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>List of CoSy engines</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>List of the selected optimisation engines that are to be integrated into DWARV</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Default and recommended option values for the loopunroll engine</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Directives that can be used with loop pragmas</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Default and recommended option values for the loophoist engine</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Default and recommended option values for the loopinvariant engine</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>Default and recommended option values for the loopscalar engine</td>
<td>69</td>
</tr>
<tr>
<td>5.6</td>
<td>Default and recommended option values for the cache engine</td>
<td>69</td>
</tr>
<tr>
<td>5.7</td>
<td>Default and recommended option values for the algebraic engine</td>
<td>71</td>
</tr>
</tbody>
</table>
First, I would like to thank Koen for the guidance and advice he gave me during the entire project and for the opportunity to work in the Computer Engineering group. Without all that, this thesis would not have been possible.

I also want to thank Razvan and Vlad for their assistance and advice. The help the two of you gave me in understanding DWARV has been invaluable in getting this thesis finished.

Further acknowledgements go to my roommates Hamid and Olivier. Thank you for sharing the office, the conversations we had and the help you gave me.

Special thanks go to my parents. They have encouraged and supported me not just during my graduation project, but during all my studies at Delft University of Technology. Without them, all this would not have been possible.

I would like to thank Arjan, Arnaldo, Demid, Eef and Erik for the company they provided me with and for the conversations we had during lunch.

I also want to mention Alexander, Anton, Gustavo, Ravi and Trivik. Thank you guys for the friendship you have offered me here in Delft.

Finally, I thank the Lord Jesus for keeping me sane during the last couple of years.

Marcel Slotema
Delft, The Netherlands
September 17, 2012
This Chapter will introduce the topic of this thesis, as well as a brief description of reconfigurable computing and hardware generation. First, Section 1.1 introduces the underlying problem that this thesis tackles. Then Section 1.2 explains how this thesis will work towards solving that problem and finally Section 1.3 will give an overview of the structure of this thesis.

1.1 Problem description

Reconfigurable computing Currently, research is done on how to efficiently use heterogeneous multicore platforms. One particular area of research is reconfigurable computing. The idea behind reconfigurable computing is to execute computational intensive parts of an application on a reconfigurable device, such as for example a Field Programmable Gate Array (FPGA). This technique is also known as hardware acceleration. The use of a reconfigurable device closes the gap between a full software application, which comes with a high flexibility, and an Application Specific Integrated Circuit (ASIC) design, which has a high performance. An application benefits from the high performance of a chunk of hardware, while maintaining the flexibility of a programmable device.

Hardware generation In order to use of a hardware accelerator on a reconfigurable device, part of the original application has to be translated into a Hardware Description Language (HDL) so that the device can be configured properly. This translation can be done manually, but that process is time consuming and requires knowledge of hardware design. Another way to translate part of the application is by using a specialised compiler, such as the Delft Workbench Automated Reconfigurable VHDL generator (DWARV) developed at Delft University of Technology. These kinds of tools generate a design in less time than it takes to do a manual implementation and are also usable without knowledge of hardware design. The drawback is that a manual design generally is more efficient.

However, there is still room for improvement of DWARV, the tool which this thesis will focus on. For example, a C function determining the sum of absolute differences was compiled with DWARV and simulated. The generated VHDL design required 254 cycles to complete. After heavy modifications of the C code, including loop unrolling and rewriting the `abs` function, a design was generated that needed only 78 cycles. By rewriting the C code, a speedup of 3.3 was achieved. This thesis will work on improving DWARV to decrease the difference between ‘original’ code and modified code.
1.2 Thesis goals

The goal of this thesis is to reduce the need for manual modifications of source code by introducing several optimisation techniques with DWARV. This goal will be pursued by first selecting a number of promising optimisation techniques available for the framework on which DWARV is based. Afterwards, these techniques will then be integrated with DWARV and finally they will be evaluated.

A hardware design can be tested on different levels, such as simulation, post-synthesis and on an actual device. This thesis will evaluate designs on simulation-level only. While the effects of delay paths and the frequency are not taken into account on this level, simulation of a design provides enough information to determine the effectiveness of optimisations. More detailed post-synthesis evaluations are not possible due to the timing constraints of this thesis.

1.3 Thesis organisation

This thesis is organised as follows: Chapter 2 will discuss research that is relevant to the work of this thesis. A selection of optimisation techniques will be made in Chapter 3. In Chapter 4 a preliminary analysis is done on the selected optimisations. Chapter 5 discusses how the optimisations are best integrated with DWARV. The results of that integration are discussed and analysed in Chapter 6. This thesis is concluded in Chapter 7.
Related Work

2.1 Introduction

The focus of this thesis is to introduce code optimisation techniques into the DWARV compiler. This Chapter will contain a survey of related literature, focusing on two subjects that are of interest for this thesis.

Section 2.2 will provide an overview of tools similar to DWARV and will conclude with a comparison between those tools and DWARV. The reason for that discussion is to provide a broader view of hardware compilation. Code optimisation techniques improve results, but other factors, such as the architecture and general approach to hardware compilation, will also affect the execution time of a design. This thesis will introduce a number of optimisation techniques. The order in which these techniques are applied influences the effectiveness of those optimisation techniques. Section 2.3 discusses different methods that can be used to arrive at a good order for the optimisations that will be introduced in this thesis. Section 2.4 will conclude this Chapter.

2.2 Hardware Compilers

The reconfigurable computing paradigm provides new opportunities to decrease the execution time of a program. This is done by transforming the computational intensive part of a program into hardware, which can be programmed on e.g. a Field Programmable Gate Array (FPGA). The main advantage of this approach is that where software performs computations one at a time, a block of hardware can execute many computations at the same time, thus speeding up the program. When using programmable devices, each program uses a customised hardware block. To make this technique accessible, software developers need to be able to produce synthesizable hardware. Since most developers are not familiar with Hardware Description Languages (HDLs), like for example VHDL, a tool is necessary to perform the conversion between a High-Level Language (HLL) and an HDL.

This thesis is primarily concerned with DWARV. DWARV is a HLL-to-HDL compiler developed at Delft University of Technology. However, DWARV is not the only hardware generator that is available. This section discusses other tools with a similar functionality. This will not be a full comparison of all available tools, but will be limited to some of the more recent developments. A more detailed comparison of available tools is found in e.g. [23].

The tools that will be discussed in this section are:

- GCC2Verilog
- The Intermediate Predicate Format
• LegUp

• A VEX to Handel-C translator

To evaluate these different tools, some criteria are outlined in Section 2.2.1. Sections 2.2.2 through Section 2.2.5 discuss each of the selected tools, based on the criteria mentioned in Section 2.2.1. Finally, Section 2.2.6 will contain a quick summary of the evaluation.

2.2.1 Criteria used for evaluation

One of the main motivations for the development of hardware generators is to close the gap between software and hardware development. To improve the efficiency of the development of a reconfigurable computing system, tools that can transform a traditional HLL, like for example C, into an HDL are required.

Many different criteria for evaluating HLL-to-HDL compilers can be used. A criterion could for example be the execution time of the generated design, the efficiency of the usage of the available resources or the performance of the resulting design. The criteria used in this thesis are based on the viewpoint of the software developer. For developers who wish to use hardware to accelerate their program, the generation of hardware should be as easy as possible. The following criteria were selected from [23]:

• The required knowledge of hardware design should be minimised: Most software developers are unfamiliar with designing hardware. If extensive knowledge of hardware design is required to use a tool properly, its usefulness to software developers will be limited. On the contrast, if a tool requires no knowledge of hardware design, it will be very useful for software developers since it will be more intuitive to use. Some examples of hardware knowledge common for HLL-to-HDL compilers are knowledge on the available resources or knowledge on how to write your code so it can be translated optimally.

• The HLL used for input should have as few modifications as possible: When an input language requires more information than normally provided by a HLL, the input language essentially becomes a new language which developers have to learn. This additional information could be, for example, which parts of the code are parallelizable or information on timing issues. Obviously, the fewer modifications of the language that are required for a tool, the more useful it is.

• The HLL used for input should have as few restrictions as possible: Not all functionality that is provided by a HLL can easily be mapped into hardware. Some examples of such functionality from the C language are pointers, function pointers and function calls. Because hardware generation for this kind of functionality is difficult, many tools do not accept the full HLL, but only a subset. The drawback is that developers cannot make use of the full functionality of the HLL. Thus, the fewer restrictions a tool has, the easier it is to use that tool.
2.2. HARDWARE COMPILERS

2.2.2 GCC2Verilog

In [13] a tool is described which translates C code into Verilog. This tool uses the GCC[14] front-end to translate the C code into the Intermediate Representation (IR) format used by GCC. A customised back-end then translates the IR into Verilog. The focus of the paper was to provide support for the entire C language. Therefore, the compiler does not yet make use of any optimisation techniques to extract parallelism other than scheduling and the techniques already built into GCC.

A Verilog design generated by the compiler consists of a datapath and a Finite State Machine (FSM). The back-end translates each of the instructions from the GCC IR into a part of the datapath and a part of the FSM, adding a new state whenever it is necessary. The FSM controls the execution flow of the hardware depending on data dependencies and conditional statements.

To support functionality that is restricted in most similar tools, like for example pointers to the memory and function calls, GCC2Verilog uses a single address space for both the software and hardware. This reduces the complexity of selecting a function for hardware acceleration, but introduces the restriction that software and hardware cannot execute concurrently in order to preserve the coherency of the memory.

To support the use of pointers between software and hardware, the address that a pointer is pointing to is inserted into the Verilog design after the software-part of the program is linked. Function calls from hardware to software are handled similar. Dynamic pointers are supported through the use of an abstracted address. The General Purpose Processor can be interrupted to translate such an abstracted address into to a real memory address. Calls from software to hardware can again be handled in a similar fashion. As a result of this approach, the processor and hardware-accelerators cannot run concurrently.

**Hardware design knowledge:** The GCC2Verilog compiler requires little knowledge of hardware design. The tool can even be used without any knowledge of hardware design. However, chances are that such a design will not improve the original design. One reason is the fact that the user is responsible for identifying a function which is to be translated into Verilog. This requires some knowledge on which functions are suitable for hardware implementation and which are not. On top of that, due to the lack of hardware and scheduling optimisation, the performance of a design depends heavily on the coding style.

**Language modifications:** Since the Verilog generator works with the GCC front-end, no special instructions of any kind are introduced.

**Language restrictions:** One of the goals of this paper was to introduce an HLL-to-HDL compiler which can support the entire set of C statements. Therefore, GCC2Verilog was created to impose no restrictions on the C code. However, this wide support comes with a potential lower execution time.
2.2.3 High Level Synthesis using the Intermediate Predicate Format

In [8], and more recently in [9], a High-Level Synthesis (HLS) framework is proposed. Where an HLL-to-HDL compiler is focused on translating given HLL code, an HLS tool combines a software part and a hardware part into a single system. Part of an HLS tool is translating HLL code into an HDL and part is modifying the code in such a way that the hardware functions are called instead of the software versions. Some HLS tools also provide support for partitioning a program into a hardware part and a software part.

The authors of the papers developed an Ada to VHDL compiler, based on logic programming. The front-end of the compiler translates the input ADA code into the patented Intermediate Predicate Format (IPF). This IPF consists of a set of formulae describing the program. For example, a formula \( \text{predecessor}(\text{op1}, \text{op2}) \) would return \( \text{true} \) if operation \( \text{op1} \) is a predecessor of operation \( \text{op2} \). The back-end of the compiler then uses a set of rules to translate these formulae into synthesizable VHDL.

Scheduling is done by first generating a general schedule. This schedule is then optimised to extract parallelism. All operations from the next state are absorbed into the current state, as long as there are no dependencies with any operation in the current state. If there is a dependency, a new state is created and becomes the new current state.

The generated VHDL is enclosed in a wrapper with a signal indicating when a hardware block should start and another signal to indicate that a hardware block has finished its execution, as well as signals for I/O.

**Hardware design knowledge:** The tool allows for the user to specify what kind of hardware model to use. At the moment, two models are supported. Both models use a FSM but the user is able to choose between a single datapath for each hardware module or a large number of parallel functional units. The former has a simpler FSM, while the later allows for more resource sharing. In order to make the best choice for a given application, the user should know the drawbacks and advantages of each model and how they fit into their application.

The user should also make a choice on the memory organisation. The tool allows data objects to be stored either embedded in a hardware module or in an external shared memory. Since the memory organisation has a huge influence on the performance, the user should know about the advantages and drawbacks of the different memory organisations.

**Language modifications:** This tool requires no modifications to be made to the Ada source code to generate synthesizable VHDL code.

**Language restrictions:** No restrictions are imposed on the Ada source code. Currently, a new front-end which can accept ANSI C as input is being developed. Because of that, it is yet unknown whether or not the C-version of the tool will impose any restrictions.
2.2. HARDWARE COMPILERS

2.2.4 LegUp

LegUp\cite{7} is a new Open Source tool for HLS. It is envisioned for future versions to automate the partitioning of a program. However, the current version only profiles a program, after which the code should be partitioned manually. After partitioning, the selected functions are translated into Verilog, a wrapper is added to each function and the software function is replaced by a call to the hardware.

For the actual conversion from C to Verilog, LegUp uses the LLVM framework\cite{17}. Once the C code is translated to LLVM’s IR, several optimisations are performed. Finally, the IR instructions are scheduled and transformed into Verilog code.

Experimental results show that a pure hardware implementation generated by LegUp is comparable to a pure hardware implementation generated by a commercial tool called eXcite.

**Hardware design knowledge:** The hardware design knowledge that is required to use LegUp is kept to a minimum. In the current release, the user is still required to examine the profiling data and to select suitable functions for hardware implementation. Some knowledge of hardware design is necessary to steer the selection process, since not all functions will experience large gains when implemented in hardware. The paper did not discuss the translation from LLVM’s IR to Verilog in detail, so it is unclear what the effect of coding style is on the resulting design.

**Language modifications:** No additional constructs are required in the original code to use the tool properly.

**Language restrictions:** Though LegUp as a HLS tool accepts the full standard of C as input, the translation from C to Verilog does not. Functions that include those restrictions cannot be accelerated in hardware and have to be executed on the processor. Currently unsupported constructs are: dynamic memory allocation, floating point operations and recursive function calls.

2.2.5 The VEX VLIW compiler in combination with Handel-C

The authors of \cite{15} approach the problem of HLL-to-HDL compilation by observing similarities with compilation for a Very Long Instruction Word (VLIW) processor. Both types of compilers exploit as much parallelism as possible in a program. However, where a VLIW compiler uses the available hardware resources as a starting point, an HLL-to-HDL compiler has as goal to retrieve information on what hardware is required and how it should be used.

In the paper, the VLIW Example (VEX) compiler is used to compile the source code into assembler for a VLIW processor, exploiting the available parallelism. The resulting assembly code is then converted into Handel-C by replacing each assembler instruction with a piece of equivalent Handel-C code. This is done with a tool called AS2HCC, developed by the authors of the paper. This tool also provides a wrapper with an interface to use the hardware. The resulting Handel-C design can be transformed into synthesizable VHDL by existing tools.
Tests with a number of functions show that, using this setup, most functions achieve a speedup with respect to the full software implementations. The speedup depends on the configuration of VEX, which will be discussed in the next paragraph, and the complexity of the function. Functions with a large resource requirement are not able to achieve a good speedup. One possible reason for this was mentioned: a design with more resources requires more time to complete the calculation and has a lower clock frequency than a design using fewer resources.

**Hardware design knowledge:** The VEX compiler needs detailed information on the available resources. The number of execution units that are targeted, the issue width, but also information on the latency of certain types of operation should all be passed into the compiler. Furthermore, the configuration of the memory is also adaptable. The toolchain requires information on whether to use a single-port or a dual-port memory and on the bitwidth of the memory.

All of this information has an impact on the generated design. Therefore, the user should know what this information means and how it affects the resulting design. Thus some limited knowledge of hardware design is required.

**Language modifications:** Since all the code is parsed by the VEX compiler, this toolchain does not introduce any modifications to the C language, other than the modifications, if any, that were introduced by the VEX compiler.

**Language restrictions:** Since this toolchain combines two tools, no further restrictions are introduced. Neither does the toolchain relax the restrictions imposed by VEX, if any, and by Handel-C.

As a result, dynamic memory allocation, floating point operations are not supported.

### 2.2.6 Quick summary

Table 2.1 provides a brief summary of the tools that were discussed, as well as DWARV. As can be seen, most of these tools require some knowledge of hardware design and require no modifications to the source code. Most tools also impose some restrictions on the C code that is accepted. The IPF solution is not considered when comparing the restrictions, since it only provides support for ADA. GCC2Verilog is the only tool that accepts the full range of C code, but at the cost of a slightly reduced performance.

### 2.3 Compiler Optimisation Space Exploration

Most compilers provide numerous optimisations, aimed at improving the program they are generating. When integrating different optimisation techniques into a compiler, the following questions arise: When should optimisation \( x \) be used? And which optimisation should be applied first, second, and so on? To answer these questions, methods for optimisation space exploration have been studied. This Section will discuss different techniques to perform this optimisation space exploration.
### 2.3. COMPILER OPTIMISATION SPACE EXPLORATION

<table>
<thead>
<tr>
<th>Hardware design knowledge</th>
<th>Modifications</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWARV</td>
<td>Function identification and coding style</td>
<td>Function identification pragmas</td>
</tr>
<tr>
<td>GCC2Verilog</td>
<td>Function identification and programming style</td>
<td>None</td>
</tr>
<tr>
<td>Intermediate Predicate Format</td>
<td>Hardware models and memory organisation</td>
<td>None</td>
</tr>
<tr>
<td>LegUp</td>
<td>Function identification</td>
<td>None</td>
</tr>
<tr>
<td>VEX VLIW compiler</td>
<td>Resource information, operation latency and memory organisation</td>
<td>VEX modifications</td>
</tr>
</tbody>
</table>

In order to be useful for this thesis, the methods should meet the two following criteria:

- The method should be suitable to produce the best result over a variety of test functions. A lot of research is being done on adaptive optimisation. During the compilation of a program, the source is examined and the most suitable optimisations are selected, based on the characteristics of the code. This produces the best program for a given source. These methods are also mentioned in this Section, because they can be adapted to provide good average results for a number of programs. Without support for adaptive compilation, the optimisations in DWARV should work well on any program.

- The method should be able to provide and evaluate an ordering of the different phase optimisations. Since optimisations will be integrated with DWARV without any prior knowledge on the effect of ordering on the selected optimisation techniques, it is important that the best ordering can be found.
2.3.1 The Compiler Optimization Level Exploration framework

In [12], a framework called Compiler Optimization Level Exploration (COLE) is explained. The authors use an evolutionary programming technique to find Pareto optimal optimisation chains.

In the framework, populations of a set of optimisations (called an entity) are evaluated. For each population, the best entities are kept. These entities may then be transformed by mutations, crossovers or migrations to create a new population. This process is repeated with the new population until no more new Pareto optimal entities are found. The use of Pareto optimality allows this method to focus on more than one metric.

The authors tested their framework on the GCC compiler. They applied the technique mentioned earlier to the sixty optimisation techniques built into GCC and compared the results to the default optimisation levels (-O1, -O2, -O3 and -Os). COLE was able to find a number of optimisation chains that outperformed the GCC optimisations by 37%. For this test, execution time and compilation time were the two main metrics.

**Evaluation**  This technique can be applied to find the best optimisations for both a single program, as well as the optimisations that work well on a larger set of programs. Using this technique to find a general set of optimisations is simply a matter of writing a cost-function that takes into account all the programs that should be evaluated. Adapting this technique to determine the best phase ordering is more involved. As described in the paper, the technique works by selecting which optimisations should be enabled and which should not be enabled. As the order of optimisations is fixed in GCC, this technique gives good results. However, when it comes to determining optimisation levels for new compilers, the technique lacks the ability to evaluate the influence of the order in which the selected optimisations take place. By modifying the way mutations, crossovers and migrations are done, it should be possible to modify the strategy in such a way that the effect of ordering can be taken into account, but this has not yet been investigated.

2.3.2 Optimizing General Purpose Compiler Optimizations

The authors of [10] introduce two new approaches to optimisation space exploration and combine them together. Firstly, they introduce a cost function that takes into account the effect of optimisations on a larger set of programs. Secondly, a strategy for reducing the search space is proposed.

To investigate the cost of a set of optimisations, taking into account a set of programs, first the cost of each program is determined for each program individually. Each of these costs is then normalised with respect to the cost of the program without any optimisations. These costs are averaged to end up with a single figure describing the average cost of a set of optimisations.

The strategy used to explore the search space is divided into three steps. Step one is the search for sets of optimisations that achieve well when combined together. By augmenting each set with optimisations that are not yet used in another set, larger sets of optimisations are generated. When an optimisation $i$ is already in a set, that optimisation will not be included in other sets of optimisations. This methodology limits
2.3. COMPILER OPTIMISATION SPACE EXPLORATION

the search space, because instead of individual optimisations the larger sets are used for the space exploration.

Since the purpose of the strategy is to find a set of optimisations that work well on many different programs, the second step combines the different sets produced by step one. Two sets of optimisations can be combined if the sets have no adverse effects on each other. As with step one, once two sets are combined, neither of the original sets are candidates to be put in another set.

The first two steps result in sets of optimisations. In step three, the best of these sets is selected, based on the costs of each set and favouring sets with more optimisations enabled. This strategy was applied to GCC and resulted in a set of optimisations that performed slightly better, in the order of several percent, than the built-in optimisation sets for the SPECint95 benchmark.

Evaluation The use of the specified cost function allows this method to be used to evaluate a large number of programs. However, the drawback of this strategy is the fact that the order in which optimisations take place is not considered. Though incorporating different orderings into the strategy is possible, it would increase the number of available sets drastically. More sets result in a larger search space and a larger execution time of this strategy.

2.3.3 Machine Learning Compiler

In [2], machine learning techniques are used to assist the compiler in selecting a good set of optimisations for a program.

With machine learning, one wants to model the effectiveness of a series of optimisations, based on characteristics of the program. Using this model, a series of optimisations can be identified for a new program and perhaps even extended. Since two different programs would benefit from a different series of optimisations, this model-based approach allows the compiler to focus the search for new optimisations on what worked well for previously encountered, yet similar, programs.

To use this approach, it has to be possible to determine which programs are similar. This can be done by comparing ‘features’ of a program. Such a feature could be any property of the program. Examples are the number of loops in the program or the average number of function calls per function. The choice of these features is important for the result produced by the machine learning approach. A feature like the number of lines of code for example is irrelevant when considering code optimisations.

Evaluation This approach of machine learning is useful when determining which optimisation to apply on a single program, but the question is if it can be used to find a good series of optimisations that work well on multiple programs so that it can be used for this thesis? For different types of programs, the approach finds good series of optimisations to perform for each type of program. Several of these optimisation series can then be combined into a general series of optimisations that will work well for most programs. The effect of different orderings of the same optimisations can also be discerned by this
approach, as long as the sequence of optimisations can be changed and as long as the learning algorithm can distinguish the results of different orderings.

2.3.4 Exhaustive Optimization Phase Order Space Exploration

While the search space for compiler optimisations is generally regarded as extremely large, the authors of [16] claim that the search space is not as large as is often thought. In their paper they also propose a method to perform an exhaustive exploration of the search space.

While the number of possible optimisation sequences grows exponentially, not every sequence of optimisations produces different programs. The search space can be significantly reduced by not considering the sequence of optimisations as the goal of the search, but rather to consider the resulting program instances as the goal of the search.

The search space can then be explored by starting with the unoptimised program. To determine the program instances in the next level, all optimisations are attempted on the instances from the previous level. After this is done, all the generated instances are compared to each other and all instances that are the same are merged. This last step is the key to reducing the search space.

The major bottleneck of this technique is the comparison of two program instances. The paper proposes a method that uses three metrics: the instruction count, the byte-sum of all instructions and a cyclic-redundancy code checksum. If these three metrics are equal for two instances, the instances are considered to be equivalent. Differences in the allocation of registers can be taken into account by renaming all registers, starting from R1 for the first register and so on.

Evaluation This technique is geared towards optimising a single program. While this technique can be applied on many different kernels, it results in the best sequence of optimisations for each program individually. Determining the most optimal sequence of optimisations for all kernels requires analysing all program instances for all kernels. Based upon that analysis, the best average sequence of optimisations can be determined. However, due to the amount of data that will require analysis, this approach does not seem feasible. This approach is usable to determine the ordering in which optimisations are applied. If two different sequences of optimisations lead to the same program instance, the two sequences will be merged.

2.3.5 Compiler Optimization-Space Exploration

The authors of [22] discuss a method to reduce the search space in order to efficiently perform iterative compilation on each program. Iterative compilation is a technique where source code is compiled with an initial set of e.g. optimisations, the result is evaluated and based on previous results a new set is determined.

The proposed method to reduce the search space focuses on a single level of the search tree before advancing to the next level. For the first level, the source code of a program is compiled with each of the available optimisations enabled. The resulting programs are then compared with each other and the best performing optimisation is selected. With
2.4. CONCLUSION

the first optimisation selected, the next level applies a similar procedure to determine
the second optimisation in a chain. The resulting programs that are evaluated are the
result of first applying the first optimisation and then applying all other optimisations.
Suppose there are \( k \) different optimisations and \( n \) optimisations are chained. A full
exhaustive exploration of the search tree would require \( k^n \) compilations. By selecting
the best optimisation one by one, only \( k \cdot n \) compilations are required.

By selecting ‘the best’ optimisations at each level and discarding all other optimi-
sations, this method does not take full advantage of interactions between different
optimisations. For example, suppose there is an optimisation that does not boost the
performance significantly on itself. Also suppose that there is another optimisation that
provides a major increase in performance after the first optimisation is performed. The
first optimisation will never be selected by this method and thus the major increase in
performance provided by the second optimisation will not be achieved.

**Evaluation** This method is designed to work on a single program, but not on a larger
set of programs. Applying this method on a set of programs will cause problems. If
averaging of the performance is done at each level, only optimisations that work well on
all programs will be selected. Optimisations that work well on one type of program, but
not on others may be disregarded. Thus this method may not result in the best set of
optimisations. The advantage of this method is that it is not just focused on enabling
or disabling optimisations, but that it will result in a sequence of optimisations.

2.3.6 Quick summary

All approaches discussed in this Section will require modifications if they are to meet
the criteria outlined in the beginning of this Section. The Compiler Optimization-Space
Exploration produces results of questionable quality for this thesis and is therefore not
considered suitable. The Exhaustive Optimization Phase Order Space Exploration ap-
proach requires large modifications and the Optimizing General Purpose Compiler Opti-
mizations is likely to require too much runtime. That leaves the COLE and the machine
learning approaches. The machine learning approach requires fewer modifications and is
the best approach to optimisation space exploration for this thesis.

2.4 Conclusion

This Chapter has discussed several recent HLL-to-HDL compilers. Without experiments,
it is impossible to determine which compiler is best. However, based on the comparison
done in this Chapter, DWARV is comparable to other recently developed tools. Out of
all the mentioned compilers, LegUp is most similar to DWARV. Both of them share ap-
proximately the same restrictions and required knowledge. Furthermore, the architecture
targeted by LegUp is comparable to the MOLEN architecture, targeted by DWARV.

Also discussed in this Chapter were a number of approaches for optimisation space
exploration. None of the approaches are suitable for this thesis without modifications.
Most useful are the cost function, as described in [10], and the machine learning ap-
proach of [2]. The cost function allows for a quick and fair comparison between different
optimisations, based on tests with any number of programs. The exploration method using machine learning is the easiest to adapt for this thesis and it takes into consideration that programs with different characteristics can benefit from different sequences of optimisations.
3.1 Introduction

This Chapter will introduce optimisations. It will start with a discussion on how a VHDL design can be optimised, followed by a list of possible optimisations. Finally, these two parts will be combined and a number of optimisations will be selected to integrate them with DWARV.

The Chapter will begin with a discussion on how to evaluate a VHDL design in Section 3.2. In Section 3.3, the metrics found in the previous Section will be translated into requirements for good optimisation techniques. Section 3.4 will briefly explain the structure of DWARV, after which Section 3.5 will provide a list of possible optimisation techniques offered by the CoSy framework. Out of these techniques, several will be selected in Section 3.6 based on the requirements for optimisations. Finally, Section 3.7 will briefly summarise this Chapter.

3.2 Criteria to evaluate generated VHDL

The goal of applying optimisations is to generate a VHDL design that is ‘better’ than the design that was generated without applying the optimisations. But what is a ‘good’ design? This section will list three criteria that are important for a ‘good’ design: the time it takes to execute the design, the area that is used by the design and the power that the design requires to operate. All these criteria are related to each other. For example: a design that has to execute as fast as possible is likely to be a larger design than one that is designed to run slower, while a design that is designed to operate on low power is likely to be slower than a medium or high power design.

3.2.1 Execution Time

Execution time has been a major consideration in the field of computer engineering for years. If a system A has a lower execution time when compared to a different, slower system B, system A will be able to do more work in a unit of time than system B. Hence, system A, the faster system, is more productive than system B.

The execution time of a VHDL design that is generated by DWARV consists of two factors: the number of cycles that the design requires to finish its execution and the period of a single cycle. The total execution time of the design can be expressed as the product of these two factors.

Data dependencies play an important role in the number of cycles that a design requires. If the results of an operation, say \( b + 1 \), is calculated in cycle \( n \), the result of this operation cannot be used prior to cycle \( n + 1 \). Therefore, a design with more
dependencies between data will require more cycles, and thus have less parallelism than
a design with less data dependencies. Another aspect of the number of cycles is the
latency of operations. Certain operations may require more cycles to determine a result
than others. Suppose that an integer addition has a latency of one cycle and that an
integer division has a latency of ten cycles. An operation that is dependent on the result
of the division will have to wait longer than an operation which depends on the result of
the addition. The actual latencies of operations depend on the implementation device.
For example, if the device has a dedicated divider, divisions will have a shorter latency
compared to a device which does not have such a dedicated divider.

The period of a cycle, or the clock frequency, is limited by the longest path between
two subsequent cycles. If a cycle requires a time longer than the cycle period to complete,
the correctness of the design cannot be guaranteed. The cycle period is dependent
on the technology on which a design is implemented. The size of the transistors, the
placement and routing of different blocks and all kinds of other factors play a crucial
role in determining the longest path. Because of this, the effect of changes in a design at
the IR level on the cycle period is expected to be either very small or very complicated.
Therefore, the prime focus to improve the execution time will be the reduction of the
number of cycles of a design. Throughout this thesis, the execution time of a design will
be measured in the number of cycles of a simulated run of the design.

3.2.2 Used area

The area that is used by a design is determined by its implementation technology. In
the field of reconfigurable computing, this is mostly done on Field Programmable Gate
Arrays (FPGAs). A design, usually equivalent to a software function, should be as small
as possible. This way, multiple different designs, or hardware kernels, can be placed on
a single FPGA and the program that is accelerated experiences a larger speedup.

Important factors determining the area of a design are the number of operations that
are used in a program and the locality of the uses of the variables. The toolchain that is
used to implement a design can also have a huge influence on the area. A toolchain that
is specific for a family of FPGAs creates a more area-efficient design than a toolchain
which supports a broad range of FPGAs.

Each operation that might be executed corresponds to a block that is required in the
HDL design. Therefore, the more operations there are in a program, the more blocks
are required to execute each operation. Suppose each block requires a unit of area to be
implemented. If the number of blocks is doubled, so will the area that is used. In general,
this is not a linear process, but it holds that more operations, and thus execution blocks,
will result in a larger area for a design.

The locality of the use of a variable is an indication of how close the uses of the
variable are to the assignment of the variable. If uses of a variable are spread throughout
a program, and the locality of the variable is low, this will result in a larger design. The
use of a variable has to be physically wired to its assignment. If a variable has a good
locality, short wires will suffice. But if a variable does not have a good locality, the wires
will be longer and the routing of the design will be more complicated.
3.2.3 Power considerations

The power dissipation of a system is another important factor describing the effectiveness of a system. If the power dissipation is too large with respect to the capacity of a system to dissipate the heat, a processor will not survive long. More energy is created than can be dissipated, so the net energy, which is related to the temperature, will increase. The net effect of this increase in energy is the (self-) destruction of the processor. Another reason to consider the power used by a design is systems which have to operate with a limited power supply. Examples of this class of devices are implantable medical devices, such as pacemakers or neurostimulators. However, for this thesis, the focus of power considerations is on reducing the heat generation. Reconfigurable computing requires a reconfigurable device, such as FPGAs. These devices typically consume more power than a General Purpose Processor (GPP). Because of this, the reconfigurable device is the most power-intensive part of a system and care should be taken to produce a power-efficient design.

The power used by a system can be divided into two parts: static and dynamic power. Static power is due to physical effects that occur within a physical implementation, independent of the operation that is executed. One example is the leakage current in a MOSFET. Dynamic power is due to switching activity. At the VHDL level, static power may or may not be affected. If the implementation device does not connect transistors that are not used, the static power will be proportional to the area of the design. Otherwise, the static power will be constant, independent of the design. Dynamic power can be described by the following equation

\[ P_{\text{dyn}} = nCV^2f \]

Here, \( n \) is the average switching activity, \( C \) is the capacity that will need to be (dis)charged, \( V \) is the supply voltage and \( f \) is the switching frequency. Reducing the capacity or the supply voltage is not possible through changing the VHDL description. So in order to reduce the power consumption, either the frequency or the average activity should be reduced. For a discussion on how to change the clock frequency, please read Section 3.2.1. The average activity can be reduced by scheduling fewer operations in a cycle. However, both reducing the frequency and the average activity will increase the execution time.

3.3 Requirements for optimisations

In order to assess the usefulness of optimisations, the main focus of investigation is on improving the performance in terms of execution time of a kernel. Other measures of performance include the area that is used for the design and the power that it uses (usually measured in the frequency at which a design runs). The following three requirements can be used to evaluate the usefulness of an engine.

1. Reducing the number of expensive operations. An operation can be seen as expensive when it requires more than e.g. three cycles when most operations require
only a single cycle. Examples of expensive operations are multiplications, requiring seven cycles in the current version of DWARV, and divisions, requiring sixty-nine cycles in the current version of DWARV. Depending on the implementation of the divider, removing a single division could decrease the number of cycles of the overall design by sixty-nine cycles. Constant multiplications and divisions are not expensive operations, since often specialised hardware can be generated to perform these operations.

2. Another type of operations that increases the execution time is the memory access operation. Currently, DWARV schedules four cycles for a memory access. The exact latency depends on the memory that is used. A larger memory comes with a larger latency. The memory is pipelined, so after the initial latency, one memory location is read each cycle. Each memory access introduces a high-latency dependency. As such, it is best to reduce the number of memory accesses as much as possible.

3. A common feature in kernels are loops. The execution time of a loop can be characterised by two properties. The first property is the number of iterations of the loop body. If the number of iterations is doubled, the loop will take twice as long to finish its execution. The second property is the number of cycles of the loop body. Suppose that the number of cycles of the loop body is doubled. Each iteration will then take twice as long to complete. Therefore, the execution time of the entire loop will be doubled. Thus in order to reduce the execution time of a loop, either the number of iterations or the number of cycles per iteration or both should be decreased.

3.4 Structure of DWARV

This thesis focuses on the implementation of DWARV with the CoSy framework[3], also called DWARV 2.0[19]. When translating a C function into VHDL code, DWARV goes through a number of stages: the front-end, a lowering stage, an optimisation stage, the scheduler and a code-generation stage.

The front-end is responsible for handling the input of the compiler. It parses the C code in the input file, performs syntax checking and returns the program in an Intermediate Representation (IR). Furthermore, it also checks whether or not a function should be translated. This selection can be done manually or by an external program. Either way, the information on which function to translate is passed to DWARV through special pragmas.

After the front-end, the IR is passed onto the lowering stage. This stage rewrites complex constructions in the IR, for example a switch statement, into a less complex alternative, which is in the case of a switch statement a series of if-else statements. The reason for doing this is to make the stages that follow simpler since they do not need to support the more complex constructions.

The optimisation stage rewrites the IR in a way that will produce a ‘better’ result. General optimisation techniques are applied here, as well as optimisations designed for hardware-generation. This thesis will focus on the optimisation stage.
The scheduling stage has to do several things. First, the IR should be matched against a set of rules. A rule contains instructions on how IR blocks can be translated into VHDL. The actual translation does not take place yet, but this information is necessary for the next step. After the matching comes the scheduling of all the instructions. Each instruction is scheduled to be placed inside a state within the Finite State Machine (FSM). DWARV does this using an As Soon As Possible (ASAP) algorithm. Ideally, the FSM should have as few states as possible. However, the number of instructions that can be put in the same state is limited mostly to dependencies.

After the scheduling is done, the code-generation stage is reached. During this stage, the scheduling information is transformed into a VHDL file. VHDL is generated for the interface and the state machine is generated. For each instruction that was matched, the corresponding VHDL is created.

The CoSy framework is based upon engine passes. An engine is an algorithm, written in a language like C that can be applied on the IR\[6\]. Most engines will analyse and modify the IR, but there are also more complicated engines. Such as the parser and syntax checker and the engines that assist in the code generation. Each stage that was mentioned previously consists of one or more engines. Take for example the lowering stage. There is an engine for lowering a switch statement, an engine for lowering array references, an engine for lowering so called bitfields, and so on. Many of these engines are supplied with the CoSy framework, but some are written specially with DWARV in mind.

This thesis will focus on engines for the optimisation stage. Since there are many engines supplied with CoSy that implement optimisation algorithms, no new engines will be created. After all, there is no need to reinvent the wheel.

### 3.5 Available CoSy engines

Table 3.1 lists all available CoSy engines, whether or not they optimise the code, what their priority should be and a brief description. The descriptions are based on documentation that is supplied with the CoSy framework[4,5]. Optimisation engines whose priority is marked with an * are already present in DWARV. Explanations on the priority of engines can be read in Section 3.6.

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algebraic</td>
<td>Yes</td>
<td>High</td>
<td>Performs algebraic simplifications</td>
</tr>
<tr>
<td>alias</td>
<td>No</td>
<td></td>
<td>Annotates objects with information on whether or not they might be referred to by pointers</td>
</tr>
<tr>
<td>allocpr</td>
<td>No</td>
<td></td>
<td>Determines stack offset of local variables</td>
</tr>
<tr>
<td>alloctp</td>
<td>No</td>
<td></td>
<td>Determines type sizes and record offset</td>
</tr>
<tr>
<td>anc0</td>
<td>No</td>
<td></td>
<td>C89 front-end</td>
</tr>
</tbody>
</table>
Table 3.1 (contd)

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anc0embc</td>
<td>No</td>
<td></td>
<td>C front-end with Embedded C extensions</td>
</tr>
<tr>
<td>basebind</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites global data accesses into an access to a base pointer + offset</td>
</tr>
<tr>
<td>beg</td>
<td>No</td>
<td></td>
<td>Back-end generator</td>
</tr>
<tr>
<td>blockmerge</td>
<td>Yes</td>
<td>Med*</td>
<td>Merges as many blocks together as possible</td>
</tr>
<tr>
<td>blockorder</td>
<td>Yes</td>
<td>N/A*</td>
<td>Reorders blocks to minimise the number of branches</td>
</tr>
<tr>
<td>bpredict</td>
<td>No</td>
<td></td>
<td>Annotates blocks with usage estimates</td>
</tr>
<tr>
<td>C89Front</td>
<td>No</td>
<td></td>
<td>C89 front-end</td>
</tr>
<tr>
<td>cache</td>
<td>Yes</td>
<td>High</td>
<td>Reducing memory accesses by delaying the writing to the memory as far as possible. Temporary values are used for accesses in between.</td>
</tr>
<tr>
<td>cfgedges</td>
<td>No</td>
<td></td>
<td>Repairs the control flow graph (CFG)</td>
</tr>
<tr>
<td>CFront</td>
<td>No</td>
<td></td>
<td>C front-end with C99 features</td>
</tr>
<tr>
<td>chainflow</td>
<td>Yes</td>
<td>Med</td>
<td>Removes redundant nested if-statements</td>
</tr>
<tr>
<td>checkcfg</td>
<td>No</td>
<td></td>
<td>Checks for problems in the CFG</td>
</tr>
<tr>
<td>checkmir</td>
<td>No</td>
<td></td>
<td>Checks for problems in the CoSy Common Medium Level Representation (CCMIR)</td>
</tr>
<tr>
<td>ckfstrength</td>
<td>Yes</td>
<td>None</td>
<td>Replaces standard function calls with cheaper alternatives</td>
</tr>
<tr>
<td>condassigncreate</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites assignments guarded by an if-statement into a conditional assignment</td>
</tr>
<tr>
<td>conevun</td>
<td>Yes</td>
<td>Med*</td>
<td>Performs constant evaluation and expression simplification</td>
</tr>
<tr>
<td>constprop</td>
<td>Yes</td>
<td>High</td>
<td>Performs constant propagation</td>
</tr>
<tr>
<td>copyprop</td>
<td>Yes</td>
<td>Med*</td>
<td>Performs copy propagation</td>
</tr>
<tr>
<td>cse</td>
<td>Yes</td>
<td>Low</td>
<td>Performs common subexpression elimination on regions in a procedure</td>
</tr>
<tr>
<td>cvtuuintreal</td>
<td>No</td>
<td></td>
<td>Rewrites conversions of unsigned integers to floating point numbers as a conversion between signed integers and floating point numbers.</td>
</tr>
<tr>
<td>deadblock</td>
<td>Yes</td>
<td>Not</td>
<td>Removes unreachable blocks</td>
</tr>
<tr>
<td>deadcode</td>
<td>Yes</td>
<td>Med*</td>
<td>Removes unnecessary assignments</td>
</tr>
<tr>
<td>deadobj</td>
<td>Yes</td>
<td>Med*</td>
<td>Removes unused objects</td>
</tr>
<tr>
<td>deadtype</td>
<td>No</td>
<td></td>
<td>Removes unused types. This will have no effect on the generated code.</td>
</tr>
</tbody>
</table>
### 3.5. AVAILABLE COSY ENGINES

Table 3.1 (contd)

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>No</td>
<td></td>
<td>Annotates the CCMIR with debugging information</td>
</tr>
<tr>
<td>defuse</td>
<td>No</td>
<td></td>
<td>Annotates the CCMIR with information on how often a variable is read or written</td>
</tr>
<tr>
<td>defusereport</td>
<td>No</td>
<td></td>
<td>Reports unused or undefined objects</td>
</tr>
<tr>
<td>demote</td>
<td>Yes</td>
<td>High*</td>
<td>Attempts to reduce the bitwidth of operations</td>
</tr>
<tr>
<td>dismemun</td>
<td>Yes</td>
<td>N/A*</td>
<td>Rewrites accesses to array elements or struct fields into memory accesses</td>
</tr>
<tr>
<td>domorder</td>
<td>Yes</td>
<td>Low</td>
<td>Reorders blocks based on a dominator tree heuristic</td>
</tr>
<tr>
<td>dwarf1</td>
<td>No</td>
<td></td>
<td>Generates debug information in the DWARF 1.0 format</td>
</tr>
<tr>
<td>dwarf2</td>
<td>No</td>
<td></td>
<td>Generates debug information in the DWARF 2.0 format</td>
</tr>
<tr>
<td>echo</td>
<td>No</td>
<td></td>
<td>Prints a specified message</td>
</tr>
<tr>
<td>EmbeddedCFront</td>
<td>No</td>
<td></td>
<td>Front-end supporting Embedded C and C99 features</td>
</tr>
<tr>
<td>expandsalloc</td>
<td>No</td>
<td></td>
<td>Translates statements using Variable Length Objects into a series of function calls</td>
</tr>
<tr>
<td>exprprop</td>
<td>Yes</td>
<td>Med*</td>
<td>Performs expression propagation</td>
</tr>
<tr>
<td>fix2fie</td>
<td>No</td>
<td></td>
<td>Replaces fixed point operations with calls to an emulation library</td>
</tr>
<tr>
<td>float2fle</td>
<td>No</td>
<td></td>
<td>Replaces floating point operations with calls to an emulation library</td>
</tr>
<tr>
<td>float2fpe</td>
<td>No</td>
<td></td>
<td>Replaces floating point operations with calls to an emulation library</td>
</tr>
<tr>
<td>funceval</td>
<td>Yes</td>
<td>Not</td>
<td>Evaluates function calls with constant arguments</td>
</tr>
<tr>
<td>globcse</td>
<td>Yes</td>
<td>Low*</td>
<td>Performs common subexpression elimination on an entire procedure</td>
</tr>
<tr>
<td>hwloopannotate</td>
<td>No</td>
<td></td>
<td>Annotates loops which are candidates to be transformed into hardware loops, based on specified requirements. Hardware loops are used by the software pipeliner.</td>
</tr>
<tr>
<td>hwloopcheck</td>
<td>No</td>
<td></td>
<td>Target dependent analysis on whether or not a loop can be transformed into a hardware loop</td>
</tr>
<tr>
<td>HWLOOPCompos</td>
<td>No</td>
<td>Not</td>
<td>Composite engine that combines the analysis and creation of hardware loops</td>
</tr>
<tr>
<td>Engine name</td>
<td>Optim.</td>
<td>Priority</td>
<td>Brief description</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>hwloopcreate</td>
<td>No</td>
<td>Not</td>
<td>Rewrites the annotated loops into the hardware loop statements</td>
</tr>
<tr>
<td>hwlooporder</td>
<td>No</td>
<td>Not</td>
<td>Places all the necessary basic blocks between the start and end statements of a hardware loop</td>
</tr>
<tr>
<td>i586cg</td>
<td>No</td>
<td></td>
<td>Back-end for Pentium assembly generation</td>
</tr>
<tr>
<td>ifconvert</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites assignments inside an if-statement into a conditional assignment statement</td>
</tr>
<tr>
<td>inliner</td>
<td>Yes</td>
<td>N/A*</td>
<td>Inlines procedure calls</td>
</tr>
<tr>
<td>ir2gdl</td>
<td>No</td>
<td></td>
<td>Generates a dump of the CCMIR in the Graphic Description Language (GDL) format</td>
</tr>
<tr>
<td>iterate</td>
<td>No</td>
<td></td>
<td>Provides a method of iterating engine declarations within an EDL file</td>
</tr>
<tr>
<td>locobjuse</td>
<td>No</td>
<td></td>
<td>Determines the usage estimate of local variables and parameters</td>
</tr>
<tr>
<td>loopanalysis</td>
<td>No</td>
<td></td>
<td>Annotates loops with loop information</td>
</tr>
<tr>
<td>loopbcount</td>
<td>No</td>
<td></td>
<td>Determines the execution frequency of blocks inside a loop</td>
</tr>
<tr>
<td>loopcanon</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites exotic loop constructs into loops that are more easily recognised</td>
</tr>
<tr>
<td>loopdump</td>
<td>No</td>
<td></td>
<td>Dumps the loop information</td>
</tr>
<tr>
<td>loopfuse</td>
<td>Yes</td>
<td>High</td>
<td>Rewrites two successive loops into a single loop</td>
</tr>
<tr>
<td>loopguard</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites while loops into do-while loops</td>
</tr>
<tr>
<td>loophoist</td>
<td>Yes</td>
<td>High</td>
<td>Applies loop-invariant code motion on statements</td>
</tr>
<tr>
<td>loopinvariant</td>
<td>Yes</td>
<td>High</td>
<td>Applies loop-invariant code motion on expressions</td>
</tr>
<tr>
<td>loopive</td>
<td>Yes</td>
<td>Med</td>
<td>Reduces the number of loop induction variables</td>
</tr>
<tr>
<td>looppragma</td>
<td>No</td>
<td></td>
<td>Annotates the CCMIR according to loop-specific pragmas in the source code</td>
</tr>
<tr>
<td>loopremove</td>
<td>Yes</td>
<td>Low</td>
<td>Removes loops that are empty</td>
</tr>
<tr>
<td>looprev</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites a loop such that it counts down, instead of up</td>
</tr>
<tr>
<td>loopscalar</td>
<td>Yes</td>
<td>High</td>
<td>Replaces a write to the same address in each loop iteration by a temporary variable and a write after the loop</td>
</tr>
</tbody>
</table>
### Table 3.1 (contd)

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loopstrength</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites array accesses to pointer expressions</td>
</tr>
<tr>
<td>looptest</td>
<td>Yes</td>
<td>Low</td>
<td>Attempts to simplify the loop-condition</td>
</tr>
<tr>
<td>loopunroll</td>
<td>Yes</td>
<td>High</td>
<td>Performs loop unrolling</td>
</tr>
<tr>
<td>lowerbitfield</td>
<td>Yes</td>
<td></td>
<td>Rewrites bitfield operations into logical operators</td>
</tr>
<tr>
<td>lowerboolean</td>
<td>Yes</td>
<td></td>
<td>Rewrites booleans into integers</td>
</tr>
<tr>
<td>lowerboolval</td>
<td>Yes</td>
<td></td>
<td>Rewrites a boolean that is to be stored as an integer into a an assignment using a temporary variable</td>
</tr>
<tr>
<td>lowercirc</td>
<td>No</td>
<td></td>
<td>Rewrites circular pointer arithmetic into calls to an emulation library</td>
</tr>
<tr>
<td>lowercom</td>
<td>Yes</td>
<td></td>
<td>Lowering of initialisers of structures, arrays or unions</td>
</tr>
<tr>
<td>lowercondassign</td>
<td>Yes</td>
<td></td>
<td>Rewrites conditional assignments into if statements</td>
</tr>
<tr>
<td>lowerfix</td>
<td>No</td>
<td></td>
<td>Rewrite fixed point arithmetic into calls to an emulation library</td>
</tr>
<tr>
<td>lowerhwloop</td>
<td>Yes</td>
<td></td>
<td>Rewrites hardware loops with special annotation into a regular loop</td>
</tr>
<tr>
<td>lowerintconst</td>
<td>Yes</td>
<td></td>
<td>Rewrites ‘expensive’ integer constants into fake function calls</td>
</tr>
<tr>
<td>lowermaa</td>
<td>Yes</td>
<td></td>
<td>Rewrites array assignments</td>
</tr>
<tr>
<td>lowermirright</td>
<td>Yes</td>
<td></td>
<td>Rewrites right-hand side expressions</td>
</tr>
<tr>
<td>lowerpfc</td>
<td>Yes</td>
<td></td>
<td>Rewrites pure function calls to regular function calls</td>
</tr>
<tr>
<td>lowerselect</td>
<td>Yes</td>
<td></td>
<td>Rewrites conditional assignments into if-else-statements using temporary variables</td>
</tr>
<tr>
<td>lowerstarg</td>
<td>Yes</td>
<td></td>
<td>Rewrites a struct as function argument into the passing of a pointer to the struct</td>
</tr>
<tr>
<td>lowerstret</td>
<td>Yes</td>
<td></td>
<td>Rewrites functions that return a struct to pass that struct as argument</td>
</tr>
<tr>
<td>lowervarargs</td>
<td>Yes</td>
<td></td>
<td>Rewrites functions with a variable number of arguments into all arguments passed in an array (much like the arg and argv structure of the main function in C)</td>
</tr>
<tr>
<td>lrrename</td>
<td>Yes</td>
<td>High</td>
<td>Rewrites a variable with separate life ranges into two different variables</td>
</tr>
<tr>
<td>markconvert</td>
<td>No</td>
<td></td>
<td>Marks a conversion where the variable is already in range of its target type</td>
</tr>
</tbody>
</table>
### Table 3.1 (contd)

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>misc</td>
<td>Yes</td>
<td>Low</td>
<td>Performs several basic optimisations</td>
</tr>
<tr>
<td>mkcontainers</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites a bitfield reference into a normal integer</td>
</tr>
<tr>
<td>mkmultires</td>
<td>Yes</td>
<td>Not</td>
<td>Marks functions that will return a structure</td>
</tr>
<tr>
<td>mktuples</td>
<td>Yes</td>
<td>Not</td>
<td>Wrapper for the <code>pragmatuple</code>, <code>mkmultires</code> and <code>scalarreplace</code> engines</td>
</tr>
<tr>
<td>mvpostop</td>
<td>Yes</td>
<td>Med</td>
<td>Places a post-operator assignment after its use, instead of creating a temporary variable</td>
</tr>
<tr>
<td>ndump</td>
<td>No</td>
<td></td>
<td>Dumps a human readable form of the CCMIR</td>
</tr>
<tr>
<td>nodebug</td>
<td>No</td>
<td></td>
<td>Removes debug information</td>
</tr>
<tr>
<td>noregun</td>
<td>No</td>
<td></td>
<td>Prohibits aliased objects from being put in a register</td>
</tr>
<tr>
<td>noreturn</td>
<td>Yes</td>
<td>Not</td>
<td>Removes code after a function which will not return</td>
</tr>
<tr>
<td>OptCompos</td>
<td>Yes</td>
<td>Not</td>
<td>Wrapper to provide generic optimisations</td>
</tr>
<tr>
<td>OptHighCompos</td>
<td>Yes</td>
<td>Not</td>
<td>Wrapper for high level generic optimisation levels</td>
</tr>
<tr>
<td>OptLowCompos</td>
<td>Yes</td>
<td>Not</td>
<td>Wrapper for low level generic optimisation levels</td>
</tr>
<tr>
<td>pirtoc</td>
<td>No</td>
<td></td>
<td>Translates the CCMIR back into ISO-C</td>
</tr>
<tr>
<td>pprofile</td>
<td>No</td>
<td></td>
<td>Helps with performing runtime profiling</td>
</tr>
<tr>
<td>pragmabasebind</td>
<td>No</td>
<td></td>
<td>Recognises pragmas that can be used with the <code>basebind</code> engine</td>
</tr>
<tr>
<td>pragmainline</td>
<td>No</td>
<td></td>
<td>Recognises pragmas that control inlining</td>
</tr>
<tr>
<td>pragmaswitch</td>
<td>No</td>
<td></td>
<td>Recognises pragmas that can be used with switch-statements</td>
</tr>
<tr>
<td>pragmatuple</td>
<td>No</td>
<td></td>
<td>Recognises <code>pass_in_registers</code> pragma</td>
</tr>
<tr>
<td>prefix</td>
<td>No</td>
<td></td>
<td>Adds a prefix to global objects</td>
</tr>
<tr>
<td>proc2block</td>
<td>No</td>
<td></td>
<td>Returns all blocks within a procedure</td>
</tr>
<tr>
<td>procsort</td>
<td>No</td>
<td></td>
<td>Determines a depth-first sequence of procedures</td>
</tr>
<tr>
<td>ProfCompos</td>
<td>No</td>
<td></td>
<td>Works together with the <code>pprofile</code> engine</td>
</tr>
<tr>
<td>promote</td>
<td>Yes</td>
<td>Not</td>
<td>Increases the arithmetic size to a minimum</td>
</tr>
<tr>
<td>rcontab</td>
<td>Yes</td>
<td>Not</td>
<td>Rewrites references to real constants into a constant table</td>
</tr>
</tbody>
</table>
### 3.5. AVAILABLE COSY ENGINES

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>resize</code></td>
<td>No</td>
<td></td>
<td>Helps with emulating integer arithmetic with a different size than the platform</td>
</tr>
<tr>
<td><code>respace</code></td>
<td>No</td>
<td></td>
<td>Remaps ‘spaces’, e.g. address spaces.</td>
</tr>
<tr>
<td><code>rodatal</code></td>
<td>No</td>
<td></td>
<td>Annotates global data with read-only information</td>
</tr>
<tr>
<td><code>ropars</code></td>
<td>No</td>
<td></td>
<td>Annotates parameters with read-only information</td>
</tr>
<tr>
<td><code>samplecg</code></td>
<td>No</td>
<td></td>
<td>Sample code generator</td>
</tr>
<tr>
<td><code>samplecgsinas</code></td>
<td>No</td>
<td></td>
<td>Sample code generator with inline assembly support</td>
</tr>
<tr>
<td><code>scalarreplace</code></td>
<td>Yes</td>
<td>High</td>
<td>Rewrites structures into local variables, if possible</td>
</tr>
<tr>
<td><code>setpurity</code></td>
<td>No</td>
<td></td>
<td>Determine the side effect of procedures</td>
</tr>
<tr>
<td><code>setrefobj</code></td>
<td>No</td>
<td></td>
<td>Advanced aliasing analysis</td>
</tr>
<tr>
<td><code>setrestrict</code></td>
<td>No</td>
<td></td>
<td>Handles restricted pointers</td>
</tr>
<tr>
<td><code>sharestring</code></td>
<td>Yes</td>
<td>Not</td>
<td>Merges reference to identical initialisers</td>
</tr>
<tr>
<td><code>sinasdump</code></td>
<td>No</td>
<td></td>
<td>Dumps information on inline assembly routines</td>
</tr>
<tr>
<td><code>sinasnames</code></td>
<td>No</td>
<td></td>
<td>Handles interface between ‘regular’ code and inline assembly</td>
</tr>
<tr>
<td><code>SPARCCg</code></td>
<td>No</td>
<td></td>
<td>Code generation for SPARC</td>
</tr>
<tr>
<td><code>ssta</code></td>
<td>No</td>
<td></td>
<td>Creates a new variable for each assignment (Static Single Assignment)</td>
</tr>
<tr>
<td><code>strength</code></td>
<td>Yes</td>
<td>Low</td>
<td>Applies strength reduction, replacing expensive operations with cheaper alternatives</td>
</tr>
<tr>
<td><code>swpannotate</code></td>
<td>No</td>
<td></td>
<td>Annotates loops for software pipelining</td>
</tr>
<tr>
<td><code>swploopinfo</code></td>
<td>No</td>
<td></td>
<td>Enables software pipelining for the back-end generator</td>
</tr>
<tr>
<td><code>suppragma</code></td>
<td>No</td>
<td></td>
<td>Recognises software pipelining pragmas</td>
</tr>
<tr>
<td><code>supprepare</code></td>
<td>No</td>
<td></td>
<td>Prepares code for software pipelining</td>
</tr>
<tr>
<td><code>SwpSPARCCg</code></td>
<td>No</td>
<td></td>
<td>Code generation with software pipelining for the SPARC architecture</td>
</tr>
<tr>
<td><code>swsort</code></td>
<td>No</td>
<td></td>
<td>Sorts the cases in a switch statement based on their value</td>
</tr>
<tr>
<td><code>tailmerge</code></td>
<td>Yes</td>
<td>Med</td>
<td>Merges identical ‘tail’ blocks</td>
</tr>
<tr>
<td><code>tailrec</code></td>
<td>Yes</td>
<td>Not</td>
<td>Replaces function calls within a function to the same function by a jump</td>
</tr>
<tr>
<td><code>tardesfield</code></td>
<td>No</td>
<td></td>
<td>Dumps information on the target description</td>
</tr>
<tr>
<td><code>tardesmix</code></td>
<td>No</td>
<td></td>
<td>Mix two units and target descriptions together</td>
</tr>
</tbody>
</table>
### Table 3.1 (contd)

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Optim.</th>
<th>Priority</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>No</td>
<td></td>
<td>Returns the value of the status option</td>
</tr>
<tr>
<td>uniqlabel</td>
<td>No</td>
<td></td>
<td>Creates and assigns an unique label for each variable</td>
</tr>
<tr>
<td>unit2proc</td>
<td>No</td>
<td></td>
<td>Returns all procedures in a unit</td>
</tr>
<tr>
<td>unnest</td>
<td>Yes</td>
<td>Low</td>
<td>Attempts to reduce the level of nesting for procedure calls</td>
</tr>
<tr>
<td>uswitch</td>
<td>Yes</td>
<td>N/A*</td>
<td>Lowers switch statements into if statements and jumps</td>
</tr>
<tr>
<td>utinit</td>
<td>No</td>
<td></td>
<td>Initialisation engine</td>
</tr>
<tr>
<td>vprop</td>
<td>Yes</td>
<td>Med</td>
<td>Combines constant propagation and folding while taking into account possible variable values</td>
</tr>
<tr>
<td>vshrink</td>
<td>Yes</td>
<td>Med</td>
<td>Tries to reduce the type size of local variables while taking into account possible variable values</td>
</tr>
<tr>
<td>vstrength</td>
<td>Yes</td>
<td>Med</td>
<td>Applies strength reductions while taking into account possible variable values</td>
</tr>
<tr>
<td>xref</td>
<td>No</td>
<td></td>
<td>Dumps a function call graph</td>
</tr>
</tbody>
</table>

### 3.6 Selection of optimisation engines

In this Section, the engines that were marked as being optimisation engines in Section 3.5 will be discussed in more detail. The engines will be described based upon available documentation such as [5] and prioritised according to the criteria that were outlined in Section 3.3.

The CoSy framework provides support for software pipelining through the engines from Section 3.5 that operate on hardware loops. Software pipelining is a technique that can be used to further exploit parallelism in loops. While the technique is certainly promising to be integrated into DWARV, integrating it in DWARV would require the adaptation of the scheduling engines. Because of the complexity of those engines, this thesis will focus on integrating a number of ‘standard’ engines. These engines are simpler to integrate and thus more engines can be inserted into DWARV.

#### algebraic

The algebraic engine tries firstly to optimise common factors in expressions. Secondly, it tries to balance expressions. The common factor in an expression like $ab + ac$ can be optimised by inserting parenthesis at the appropriate positions: $a(b + c)$.

In order to work, this engine requires the modulo of the result in case of overflow, as opposed to saturation of the result. To achieve this, the OVINT option is set to 1 in the target description file.

Whenever a common factor can be placed outside parenthesis, at least one multiplication is removed. Therefore, this engine satisfies the first requirement. As a side
3.6. SELECTION OF OPTIMISATION ENGINES

effect, chains of expressions are balanced by this engine. This causes an expression to be
evaluated based on a binary tree, with logarithmic depth, as opposed to a left-recursive
approach, which requires linear time. Therefore, the priority to include this engine is
high.

**basebind**  The basebind engine rewrites the addresses of all global data elements into
a base pointer plus an offset expression.

Prior to using this engine, the global data elements should be grouped. Each element
in a group shares the same base pointer.

Since this engine only changes the way addresses are determined, no expensive oper-
ations nor memory accesses are removed. Nor does this engine influences loops. On top
of that, DWARV currently provides no support for the use of global variables. Therefore,
this engine should not be investigated further.

**blockmerge**  This engine merges two basic blocks into a single block if the second block
directly follows the first block and if there is no other way to reach the second block.
After this engine has completed a run, the deadblock engine should be run to remove
unused blocks.

This engine only changes the internal control flow between basic blocks. Therefore,
it has no direct impact on the number of operations. However, this engine might have
some influence on the scheduler used by DWARV. As such, it could be possible that this
engine can reduce the number of cycles in a loop body. The effect of this engine on the
scheduler should be investigated further before this engine can be considered as a good
optimisation engine. Until said investigation is done, this engine will be considered a
medium-priority engine.

This engine is used in the current version of DWARV as part of the dolower engine.

**blockorder**  The blockorder engine changes the ordering in which basic blocks are
reached. The reordering can be done in two different modes. In the first mode, usage
estimates are used to provide a preliminary ordering. After this has been done,
blocks are reordered in such a way to minimise the number of unconditional jumps. The
second mode reorders blocks to minimise the dynamic branches.

To use this engine in the first mode, usage information should be provided by engines
such as loopbcount or pprofile. It is also advised to have run the domorder engine.

As with the blockmerge engine, this engine works on the internal ordering of blocks.
No optimisations in the area of expensive operations and memory accesses are performed.
However, the engine does influence the scheduling of operations. A loop body will still
have to execute the same number of operations, but it might be possible that this engine
will cause the operations to be executed in less cycles. The extent to which this happens
should be investigated further to determine the usefulness of this engine, based on the
posed requirements.

However, preliminary experimental results show that after disabling this engine,
DWARV is not capable of generating VHDL for a kernel containing an empty for-loop.
Therefore, either this engine or the domorder engine should be included in DWARV at
all times.
This engine is used in the current version of DWARV as part of the codegen engine.

**cache** The cache engine applies caching to all memory writes. This caching is done by assigning the value that is to be written to the memory to a temporary variable, instead of writing it back. The actual write back is delayed as long as possible. In between these operations, any read from the same memory location is replaced by a read from the temporary variable. The value of a write to the same memory location is simply stored in the temporary variable.

After running this engine, all memory accesses between the first writes and the write back operations are referring to the temporary variables. While the use of these temporary variables increases the area that is required for the design, interaction with the memory will decrease. Thus the second requirement for selection is met, so this engine receives a high priority.

**chainflow** This engine tries to optimise conditional statements. If for example a nested if statement is implied by the outer if statement, the nested if statement can be removed. Since this process can have unwanted side effects when applied to loops, the engine will skip loop constructs by default.

When an if statement is removed, the condition expression is also removed. As a consequence, any expensive operation or memory access inside that expression will also be removed. However, as this is only a side effect of the engine, it has a medium priority.

**ckfstrength** The ckfstrength engine will attempt to rewrite certain compiler known functions, or ckfs, into an alternative that will execute faster. Most ckfs can only be replaced after it has been verified that certain conditions are met. Ckfs that might be affected by this engine are: printf, str(n)cpy, memcpy, str(n)cmp, absolute value functions for complex numbers, various trigonometric, logarithmic, exponent and square root functions for complex and floating point numbers and integer min and max.

The engine does not actively try to meet any of the requirements. While the simplifications provided by this engine could potentially speed up the design, DWARV does not provide support for any library functions at the moment. Therefore, this engine should not be considered as optimisation engine for now.

**condassigncreate** The condassigncreate engine rewrites any assignment guarded by an if statement into a conditional assignment statement. A conditional assignment statement is much like the question mark and colon combination in C.

In its current state, conditional assignment statements are rewritten by the lower-condassign engine into if statements. As such, the code generator does not support these statements. If the code generator is updated to support conditional assignment statements, this engine would receive a medium priority. While it does not meet the criteria, this engine could reduce the number of if statements and have variables available at an earlier time. However, since conditional assignment statements are not supported at the moment, this engine should not be considered.
3.6. SELECTION OF OPTIMISATION ENGINES

**conevun**  The *conevun*, *conevtg* and *conevpr* engines provide constant evaluation and attempt to simplify the resulting expressions. The engines work on expressions in the entire program, type-related expressions and expressions in procedures respectively.

While these engines do not meet any of the requirements directly, the constant folding and expression simplification can reduce and simplify a significant number of operations. Operations with two constants are replaced by the result of the operation. Therefore, these engines deserve a medium priority.

The *conevpr* is currently in use in DWARV in the *dolower* engine.

**constprop**  This engine applies constant propagation to a procedure. Constant propagation involves finding variables that are assigned a constant value and replacing any use of that variable with the appropriate constant.

If one of the variables in an expensive operation is propagated to a constant, the operation may not be an expensive operation anymore. Thus this engine meets the first requirement and has a high priority.

**copyprop**  The *copyprop* engine searches for variables, e.g. a variable i, which get their value directly from other variables, e.g. variable j. Any use of the variable i is replaced by the use of variable j. The resulting design has less unnecessary variables.

None of the requirements are targeted by this engine. However, the reduction in unnecessary variables provides an opportunity to improve both the scheduling and area of the design. Fewer variables mean that the values that are needed by a statement are available at an earlier time. A design with fewer variables also requires less registers and thus area. Therefore, this engine gets a medium priority. None of the requirements are targeted by this engine. However, the reduction in unnecessary variables provides an opportunity to improve both the scheduling and area of the design. Fewer variables mean that the values that are needed by a statement are available at an earlier time. A design with less variables also requires less registers and thus area. Therefore, this engine gets a medium priority.

This engine is currently in use in DWARV his *dolower* engine.

**cse**  The *cse* engine provides common subexpression elimination. If two expressions or statements share a common expression, this engine will put in a temporary expression in place of the common subexpression. Thus the value of the subexpression will only have to be determined once. This process is applied on a subregion of a procedure.

If subexpressions contain memory accesses or other expensive operations, common subexpression elimination can reduce the number of these operations. However, since subexpressions are in general placed ‘further’ away from its use, the process of placing and routing a design becomes more complicated. Because of this, more information is necessary on the effect of this engine and of the *globcse*, which performs common subexpression elimination on an entire procedure, on the routing phase should be investigated further. Until that investigation is complete, this engine will be considered with a low priority.
**deadcode and deadobj** These engines remove unused code and unused objects. The deadcode engine searches for and removes assignments to objects that are never used afterwards. The deadobj engine is very similar, except for the fact that it searches for objects that are not referred to.

These engines remove unnecessary assignments and objects. These operations could include expensive operations and memory accesses. However, removing these operations does not change any dependencies. No other operations will be waiting for the result, according to the definition of dead code/objects. Therefore, these engines do not meet the requirements in the sense that removing these assignments and objects will have only a minimal influence on the execution time of a design. Area wise, these two engines offer an improvement by removing unnecessary registers and wires. Thus, these two engines receive a medium priority.

These two engines are used in the dolower engine of DWARV.

**deadblock** The deadblock engine removes basic blocks that cannot be reached according to the Control Flow Graph (CFG). This engine can act as a cleanup engine to be used after an engine that can change the control flow of a design.

Any removal of a basic block by this engine will not influence the execution time of a design, since any blocks that are removed are not reachable. Furthermore, since the scheduling engine uses the proc2region engine to divide a procedure based on the CFG, using this engine would not improve the scheduling of operations. Hence, this engine should not be considered for optimisation purposes.

**demote** The demote engine attempts to reduce the number of bits that are used for an operation. To determine the minimal bitwidth, the engine attempts to figure out how many bits are actually required in the design.

By reducing the bitwidth of an expensive operation, the operation can potentially be changed into a non-expensive operation. For example, a multiplication of two 32-bit integers are scheduled using seven cycles, while a multiplication of two 8-bit integers is scheduled in a single cycle. Thus, this engine has potential to reduce the number of expensive operations. As such, it receives a high priority to be included in an optimisation run.

This engine is part of the dolower engine used by DWARV.

**dismemun** The dismemun, dismemtg and dismempr engines rewrite references to a member of a structure or to an element of an array into pointer references. The engines work on expressions in the entire program, type-related expressions and expressions in procedures respectively.

These engines are a vital part of DWARV, since these are the engines that allow DWARV to support arrays and structures. Therefore, these engines are not considered optimisations and should be included in DWARV at all times.

The dismempr engine is currently used in the dolower engine.

**domorder** This engine uses a heuristic based on the concept of a dominator tree to determine a better ordering of basic blocks. A block $i$ dominates another block $j$ if block
3.6. SELECTION OF OPTIMISATION ENGINES

$j$ can only be reached through block $i$.

As with the \textit{blockorder} engine, this engine does not meet any of the requirements directly. However, since the use of this engine might improve the scheduling, it does have a low priority. It could be used as alternative to or in combination with the \textit{blockorder} engine.

\textbf{exprprop} The \textit{exprprop} engine works similar to the \textit{copyprop} engine, except for the fact that the \textit{exprprop} engine does not operate on assignments, but directly on expressions.

This engine requires information on the usage estimate of basic blocks. Therefore, an engine that determines this information, e.g. \textit{loopbcount}, \textit{bpredict} or \textit{pprofile}, should be run prior to using this engine.

While the engine does not eliminate operations, the reduction in the number of variables does provide the potential for a better design in terms of area. With fewer variables, less area is necessary for wires and registers. Therefore, this engine has a medium priority.

\textbf{funceval} When a function is called with constant arguments, most of the time its result will also be a constant. This engine evaluates these function calls and replaces the calls by the constant return value.

DWARV relies on inlining to support function calls. When a function with constant arguments is inlined, use of constant propagation engines will propagate these constants until the constant return value is determined. The use of the \textit{funceval} engine therefore provides little additional advantages and should not be used as an optimisation engine.

\textbf{globcse} The \textit{globcse} engine provides common subexpression elimination. If two expressions or statements share a common expression, this engine will put in a temporary expression in place of the common subexpression. Thus the value of the subexpression will only have to be determined once. This process is applied on an entire procedure.

See also the comments made on the \textit{cse} engine.

This engine is currently used in DWARV as part of the \textit{dolower} engine.

\textbf{ifconvert} The \textit{ifconvert} engine attempts to rewrite if statements that contain an assignment to a single variable by a conditional assignment to the variable.

While this engine does not change expensive operations or memory accesses, it does provide a potential improvement when it comes down to scheduling. An if-else-statement requires at least three separate states: one to determine the condition, one to execute the then part of the statement and one to execute the else part of the statement. A conditional assignment does not require such a complicated control flow and could be determined in a single state. However, in its current state, DWARV cannot generate valid VHDL for conditional statements. If DWARV were to be modified to support VHDL generation of conditional statements, this engine would have a medium priority in terms of the requirements. But since this is not the case, this engine is not considered to be included in an optimisation run.
inliner  This engine replaces a call to a function by the code that the function would execute. Doing this reduces the control overhead that is normally introduced by a function call in a GPP.

Since the VHDL generated by DWARV does not offer support for function calls, the use of the inliner engine is vital to provide a limited support for function calls when translating a program into a hardware design. Therefore, the inliner engine is not considered an optimisation but should always be included in DWARV.

loopcanon  The loopcanon engine can be used to rewrite specific and unusual ways of writing a loop into a loop that is more easily recognised by other engines.

The way that loops are rewritten changes nothing when considering expensive or memory operations. Also, neither the number of instruction per iteration nor the number of iterations is changed. The new, rewritten loop cannot be written into VHDL more effective than the original loop. Therefore, this engine should not be considered as an optimisation engine.

loopfuse  The loopfuse engine rewrites two consecutive loops into a single loop if this is possible. If two loops iterate the same number of times and there are no dependencies between loops, the two loops can be merged together.

While this engine does not change the number of instructions that are executed, fusing two loops together will allow the scheduler to extract more parallelism. Thus the number of loop iterations will be halved, while the average number of cycles per loop iteration will be less than twice the original number of cycles per iteration. Therefore, this engine most likely will improve the execution time of loops and should get a high priority.

loopguard  This engine rewrites a while loop that can be verified to run at least once into a do-while loop.

Transforming a while loop into a do-while loop does not change the number of iterations of a loop, not does anything change when considering the number of cycles per iteration. The only advantage would be that a do-while loop executes one test less than a while loop. Since that is an insignificant improvement, this engine should not be considered further for optimisation purposes.

loophoist and loopinvariant  The loophoist and the loopinvariant engines remove loop invariant statements resp. expressions from a loop body. The removed expressions and statements are placed right before the loop.

By placing operations that are independent of the loop iterations outside of the loop, the number of operations that is executed each iteration is reduced. If the independent operations contain expensive or memory operations, all three requirements can be met by these engines. Therefore, these engines have a high priority.

loopive  Loops with multiple induction variables are targeted by this engine. When one of these variables is redundant, the engine rewrites the use of the redundant variable into an expression using the other induction variables.
3.6. SELECTION OF OPTIMISATION ENGINES

Depending on the translation between induction variables, this engine might use more operations for each loop iteration. However, using fewer variables may also decrease the area that is used for a design and causes less routing problems. All in all, this engine receives a medium priority.

**loopremove** The *loopremove* engine removes loops that contain no statements.

This engine can reduce the number of times a loop is iterated over. However, unless a loop is emptied by an optimisation engine, it is very unlikely that kernels that are to be implemented in hardware contain an unnecessary empty loop. Therefore, this engine should be considered with a low priority.

**looprev** This engine will rewrite loops, whose body does not depend on the induction variable, into a loop that is counting down, instead of up.

What can be changed by this engine are a couple of comparisons and additions. Neither of these will have a significant impact on performance. Furthermore, there is no benefit of writing VHDL for a loop that is counting down. Thus this engine should not be considered further.

**loopscalar** The *loopscalar* engine will rewrite a loop in which every iteration a value is written to the same memory location. Instead of writing to the memory in every iteration, the value is written to a temporary variable. After the last loop iteration, the temporary variable is written back to the correct memory location.

By reducing the number of writes to the memory, this engine meets the second requirement. Apart from the reduction in memory operations, the introduction of a temporary variable is advantageous for routing and finding other sorts of parallelism. Therefore, this engine should be considered with a high priority.

**loopstrength** The *loopstrength* engine rewrites array references within a loop into expressions using pointers.

Currently, the *dismemun* is already rewriting all array references into pointers. Since the use of this engine provides no further advantage, it should not be considered to become part of the optimisation chain.

**looptest** This engine will attempt to rewrite complex loop conditions into more conventional loop conditions. It does so by checking whether or not certain constraints, depending on the actual loop condition, are met.

A rewrite of the loop condition is not likely to change an expensive operation or a memory access. If something should be read from the memory to determine a condition, the read will not be affected by this engine. Similar, if two numbers need to be multiplied, they will need to be multiplied. This engine does not change the number of iterations in a loop, nor anything in the loop body. It could possibly make the evaluation of the condition slightly more efficient, however. The effectiveness of this engine is expected to be limited, hence this engine receives a low priority.
**loopunroll**  The *loopunroll* engine performs loop unrolling. If a loop is unrolled with a factor $n$, $n$ iterations of the original loop can be completed in a single iteration of the unrolled loop.

As with the *loopfuse* engine, this engine decreases the number of iterations, but in return increases the number of operations per iteration. However, the number of *cycles* per iteration may not change proportional to the unrolling factor. This is because the new loop body may contain more parallelism due to which the body can be scheduled more efficiently. Thus, it receives a high priority.

**The lowering engines**  The lowering engines are not optimisation engines in the sense that they improve a design. Instead, they simplify operations into more simple operations. In fact, many of the lowering engines are used to provide support for the features they are lowering. For example, without the *lowerstarg* engine function calls with a structure as an argument would not be supported. Because of this the lowering engines are not considered as code improving optimisations.

**lrrename**  The *lrrename* engine rewrites variables that have two separate life ranges into two distinct variables. The life range of a variable is the ‘time’ between the assignment of a value and all uses of that value. If a variable has separate life ranges, it could be seen as separate variables.

This engine requires information that is provided by the defuse engine.

Having a variable separated into distinct variables will remove dependencies in the design. This means that a more efficient scheduling can be made. If this technique is applied to the body of a for loop, the resulting body could be scheduled in less cycles. The increase in parallelism caused by this engine makes this engine a high priority engine.

**misc**  The *misc* engine performs several minor optimisations. Included are optimisations to reduce the complexity of if and switch statements, optimising evaluation statements, processing the result of a function call and optimisations on side-effect free function calls.

This engine provides no optimisations that are related to expensive operations, memory operation or loops. If applied, the optimisations included in this engine cause minor improvements in the design. Therefore, this engine has a low priority.

**mkcontainers**  This engine rewrites assignments and reads from structure fields with a reduced bitwidth (bitfields) into special statements which specify precisely which bits should be written or read.

This engine is similar to a lowering engine. It does not produce a better result, but can be used to offer support for bitfields. As such, this engine is not considered an optimisation engine.

**mkmultires**  This engine rewrites functions which return a structure, which was defined with a special pragma, into a special structure.
3.6. SELECTION OF OPTIMISATION ENGINES

The results of this engine do not affect any of the posed requirements. In fact, the resulting structure only complicates the code generation while offering little possibility of generating more efficient VHDL. Therefore, it should not be used as an optimisation.

**mvpostop** Post operation increments (and decrements) parsed by the anc0 engine have a structure where the value of the variable is assigned to a temporary variable, the original variable is incremented and the temporary variable is used for the expression. The mvpostop engine places the increment instruction after the expression, thus eliminating the need for the temporary variable.

This engine does not operate on expensive operations, memory accesses or loops. However, the removal of a variable does provide a better resulting design. Combined with the fact that post-increments and post-decrements are fairly common in C code, this engine has a medium priority.

**noreturn** The noreturn engine removes code after a call to a function if the called function will never return control to the caller.

Since hardware cannot perform an exit() call, the only way for a function not to return control to the caller is when a called function contains an infinite loop. When that is the case, there is something wrong with the supplied code and the hardware accelerator will remain stuck in execution. Such a function should not be used in a kernel that is to be translated into hardware. Because of this, the use of the noreturn engine provides no advantages.

**promote** The promote engine is the opposite of the demote engine. It resizes expressions that are smaller than a specified size into the specified size.

The larger the size of an expression, the larger the generated hardware will be. On top of that, this engine might increase the number of expensive operations in the same way that the demote engine can decrease these operations. Thus this engine should not be considered for optimisation purposes and should not be used.

**rcontab** This engine rewrites the use of floating point constants into an access to a global constant table, where all floating point constants are stored.

This engine should be called before the dismemun engine is used.

The use of the rcontab engine introduces additional memory operations. Furthermore, global variables are not supported in DWARV. Therefore, this engine should not be used.

**scalarreplace** This engine will replace the local use of structures by the use of separate variables for each member of the structure.

References to structures are rewritten by the dismemun engine into pointer expressions and corresponding memory accesses. By using this engine before the dismemun engine, structures are rewritten to variables as much as possible and all remaining structures can be lowered by the dismemun engine. Doing this will reduce the number of memory accesses in a design that uses structures. As such it should be considered a high-priority optimisation.
Currently the `scalarreplace` engine is a part of the `dolower` engine used in DWARV. However, it is placed after the `dismempr` engine. In its current location, the `scalarreplace` engine will not be able to optimise anything, since all structures are already lowered. It is suggested to move the `scalarreplace` engine prior to the `dismempr` engine.

**sharestring**  The `sharestring` engine attempts to find constant initialisations that are the same in size and in value. When the same initialisations are found more than once, a rewrite takes place to have all initialisations use the same reference to the constant.

DWARV handles initialisations by assigning the values of the initialisation directly to signals. Since this does not work when an initialisation is done via a memory reference, this engine will only complicate the code generation instead of allowing it to generate a more efficient design. Thus this engine should not be used.

**strength**  The `strength` engine applies strength reduction techniques. For example, a multiplication with a constant power of two can be replaced by a shift operation. These reductions work mostly on operations with a constant.

While this engine can optimise expensive operations that have a constant argument, the generated VHDL is already aware of the difference between e.g. a constant multiplication and a normal multiplication. These strength reductions are probably also done by the synthesiser. Therefore, the resulting design will not benefit much by using this engine. Because of this, the `strength` engine has a low priority.

**tailmerge**  If two different execution ‘tails’ contain equivalent basic blocks, the `tailmerge` engine merges the two blocks into a single block and changes the CFG accordingly.

The merging of two tails decreases the amount of unnecessary duplicate code. While it has no effect on the operations that are executed, and thus the execution time, less code roughly translates to less area that is required for the design. Therefore, this engine has a medium priority.

**tailrec**  If a function ends with a call to itself, the `tailrec` engine will rewrite the ending function call to a jump after putting all arguments in the right place.

DWARV does not support recursive calls in its current form. Therefore, this engine cannot be used as an optimisation engine. However, this engine might be used to provide limited support for recursive function calls.

**unnest**  The `unnest` engine checks whether or not a function is dependent on or required for its parent. If this is not the case, the function is moved outside of the parent function.

In general it does not matter for VHDL generation where a function is placed. The code needs to be executed anyway. However, when a nested function can be moved out of its parent function inside a loop body, the function may be taken out of the loop body when loop invariant code motion is applied. As such, it can decrease the number of cycles that are required for a loop iteration. However, the combination of the `inline` engine and loop-invariant code motion will have the same effect. Consequently, this engine receives a low priority.
3.7. CONCLUSION

uswitch  This engine rewrites a switch statement into a corresponding series of if-else statements.

This engine is currently used in DWARV his dolower engine. Without this engine, DWARV cannot support switch statements. Thus this engine is vital to DWARV and is not considered an optimisation.

vprop  This vprop engine provides constant folding and propagation, based on the range of values a variable can have.

The use of the range of values a variable can have might introduce additional constant optimisations when compared to the regular constprop and copyprop engines. Thus this engine is also considered a medium-priority engine.

vshrink  The vshrink engine applies the same bitwidth reduction as the demote engine, but taking into account the possible value range that variables can take.

As with the demote engine, a reduced bitwidth will result in a smaller design. Reducing the bitwidth based on the value range will possibly expose more reductions than the demote engine can find. Therefore, this engine has a medium priority.

vstrength  The vstrength engine applies information on the value range to determine whether strength reduction can take place.

The advantage of this engine is similar to those of the strength engine. But since it is not likely that a synthesiser will perform an analysis of the value range, this engine can introduce rewrite operations into more efficient operations that cannot be found by the synthesiser. Thus is has a medium priority.

3.7 Conclusion

This Chapter started with a discussion of criteria, which can be used to properly evaluate a VHDL design. These criteria led to three requirements to select optimisation engines.

From the list of available engines, the optimisation engines were prioritised, based on those requirements. The list of engine that will be integrated into DWARV is shown in Table 3.2.
Table 3.2: List of the selected optimisation engines that are to be integrated into DWARV

<table>
<thead>
<tr>
<th>Engine name</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algebraic</td>
<td>Performs algebraic simplifications</td>
</tr>
<tr>
<td>cache</td>
<td>Reducing memory accesses by delaying the writing to the memory</td>
</tr>
<tr>
<td>constprop</td>
<td>Performs constant propagation</td>
</tr>
<tr>
<td>demote</td>
<td>Reduces the bitwidth of operations</td>
</tr>
<tr>
<td>loopfuse</td>
<td>Rewrites two successive loops into a single loop</td>
</tr>
<tr>
<td>loophoist</td>
<td>Applies loop-invariant code motion on statements</td>
</tr>
<tr>
<td>loopinevariant</td>
<td>Applies loop-invariant code motion on expressions</td>
</tr>
<tr>
<td>loopscalar</td>
<td>Reduces memory accesses by delaying the writing to the memory</td>
</tr>
<tr>
<td>loopunroll</td>
<td>Performs loop unrolling</td>
</tr>
<tr>
<td>lrrename</td>
<td>Rewrites a variable with separate life ranges into different variables</td>
</tr>
<tr>
<td>scalarreplace</td>
<td>Rewrites structures into local variables</td>
</tr>
</tbody>
</table>
4.1 Introduction

In this Chapter, the selected engines, see Table 3.2, will be tested one by one. These tests should give an impression on what it takes to properly integrate the selected engines into DWAIRV. On top of that do these tests provide insight into which optimisations are promising and which optimisations are not.

To test a C-to-VHDL compiler, a set of functions, also called kernels, written in C should be used. Section 4.2 will describe the kernels that will be used throughout this thesis. The optimisation engines are placed into DWAIRV for the first time and afterwards the results are evaluated in 4.3. Based upon those results, a more thorough study is made of the effect that engines have on each other. The results of that study are presented in Section 4.4. This Chapter will be closed with a short summary of the conclusions of this Chapter in Section 4.5.

4.2 Kernels used for evaluation

The kernels described in this Section were selected for a number of reasons. First, with the exception of the loop kernel, all of these kernels are extracted from existing programs. This is important for a good evaluation of optimisations. Specialised kernels can be designed in such a way that they provide ample opportunities to be optimised, but the use of such kernels in a real-life situation is very limited. Another reason for choosing these kernels was the presence of loops in them. Almost half of the selected optimisation engines specifically target loops. Finally, it was possible for these kernels to generate real test data. While the input data may not make a difference for some kernels, for other kernels, especially control intensive kernels, the input data can impact the execution time. Using real test data mimics the situation where the kernels are used in practise.

bellmanford The bellmanford kernel applies the Bellman-Ford algorithm for determining the shortest paths of a graph from a source node to all other nodes.

The kernel contains four for-loops, one of which is nested in another loop. One of these loops would benefit from loop unrolling, the other contain too much conditional code to benefit from loop unrolling.

In the tests, this kernel is run with a graph containing 15 nodes and 21 edges. All loops iterate over one of these values, so unrolling loops with a large factor such as 32 would not cause a speedup.
**count_alive** The `count_alive` kernel was extracted from a Game of Life program. It loops through a three by three grid and counts the number of neighbours that are marked.

The kernel uses a three-iteration for-loop, with another three-iteration for-loop nested inside it. With loop unrolling, these two loops could be fully unrolled. The kernel also contains loop invariant code.

As input, a simple three-by-three field is used, but for the execution of the kernel, the size of the input field does not matter.

**evolve_hw** This kernel is another kernel taken from a Game of Life implementation. The kernel runs through each element of a field, determines how many neighbours are 'alive' and updates the element with its new state.

The kernel uses a nested for-loop to visit all elements in the two-dimensional field. These two loops can be unrolled, but the amount of parallelism is limited due to conditional code in the inner loop. The kernel also contains loop invariant code and large expressions, which can be balanced by the algebraic engine.

The input for this kernel is a 15-by-15 pattern. This means that each loop mentioned before has 15 iterations. Therefore, unrolling these loops with a larger factor will not show any effect in the tests.

**fft** The `fft` kernel computes the Fast Fourier Transform of its input. It does so by using a Butterfly computation.

The kernel consists of a while loop and two nested for-loops. Possibilities for optimisation are constant propagation and loop unrolling.

The kernel uses 64 points as input, so unrolling beyond that number will not be noticeable in any test that is run with this kernel.

**fir** The `fir` kernel applies a single time step of a Finite Impulse Response filter on its input.

It contains a single for-loop, which multiplies delayed values with coefficients and sums them all up. Possibilities for optimisation are caching of memory accesses and loop unrolling.

The kernel is tested with 16 coefficients, so unrolling with a factor larger than 16 will not have an effect.

**floydwarshall** The `floydwarshall` kernel determines, just as the `bellmanford` kernel, the shortest paths of a graph.

The kernel uses five for-loops, which can be unrolled. Further features include loop invariant code and the possibility for caching memory accesses.

Just like the `boyermoore` kernel, the `floydwarshall` kernel operates on a graph containing 15 nodes. This will limit effective unrolling factors to that number.

**floatPoint_IIR7** The `floatPoint_IIR7` kernel implements a seven stage Infinite Impulse Response filter and operates on floating point numbers. The structure is very similar to the `fir` kernel, except for the fact that it contains more operations to calculate
the IIR, instead of the FIR, and the fact that the fir kernel operates on fixed point numbers, implemented as integers, while the floatPoint_IIR7 kernel operates on single-precision floating point numbers. The possibilities for optimisation are roughly the same because of the similarities between the kernels.

The input data used for this kernel consists of 256 points.

**FracShift** The FracShift kernel shifts the elements in an array a number of times and for each shift determines the weighted sum of all elements. This is done through the use a regular pattern of arithmetic operations, structured by the use of some for-loops.

The kernel contains five for-loops. The lack of conditional code inside there loops makes them a good candidate for loop unrolling. On top of that, the kernel makes heavy use of arrays, leading to good possibilities for software caching.

All the loops within the FracShift kernel have a constant number of iteration.

**grid_iterate** The grid_iterate kernel iterates over a three dimensional grid and sets each element in the grid to either one or zero or the average of all the element’s neighbours.

Caching and loop unrolling provide possibilities to optimise this kernel, as well as some minor loop invariant code motion and common subexpression elimination.

The kernel uses a 32 by 64 by 16 grid. High unrolling factors will make little difference when compared to unrolling factors with the values of the grid size.

**hamming** The hamming kernel performs a bitwise exclusive or on two short integers and counts the number of ones in the result of the exclusive or.

This kernel uses a single for-loop that iterates over all the bits of a short integer. The possibilities for optimising this kernel are limited to applying loop unrolling on the main for-loop.

Since this kernel always operates on short integers, the actual input data will not influence the execution time.

**hw_non_max_supp** The hw_non_max_supp kernel is part of a canny edge detection program. The kernel takes the gradient of an image as input. In that gradient image, all the values that are not near a maximum are suppressed. The result is that the edges, or maxima in the gradient image, will stand out from the rest of the image.

The kernel uses four for-loops. Two of those are used for initialisation of arrays. The other two are nested and loop through rows and columns of an image. Loop unrolling can be used to improve the initialisation loops, but the inner loop of the two nested loops contains much conditional code, so the parallelism that is exposed will be limited.

As input, a 32-by-32 image is used. Generating input data for large images, such as for example a 64-by-64 image was not possible, due to memory restrictions.

**loop** The loop kernel calculates the sum of all elements of an array in a linear fashion. This is achieved using a for-loop.

Possibilities for optimisation are few, due to the simplicity of the kernel. This kernel is a good candidate for loop unrolling.
The kernel is hard-coded to work on arrays with a size of 32. The simplicity of this kernel makes it less suitable for testing how DWARV performs with useful kernels.

**mandelbrot** The *mandelbrot* kernel determines a Mandelbrot set, based on a specified resolution and the number of iterations.

The kernel uses two for-loops to go through all x and y coordinates. Inside those loops is a while-loop. The two for-loops can be unrolled, but the while loop is not suitable for unrolling. The other possible optimisation of this kernel is loop invariant code motion.

A Mandelbrot set is generated for a 150-by-150 image. The set is refined in 50 iterations of the algorithm.

**multiply** The *multiply* kernel multiplies two arrays of characters into a matrix-like array. Internally, the character arrays are first transformed into short integers, after which they are multiplied.

The kernel contains three loops, one to set the resulting array to zero and two to iterate over all the array elements. All three of these loops can benefit from unrolling them. The kernel does not contain loop-invariant code, nor many memory accesses.

The kernel is tested with an input size of 32. Since all loops will run 32 iterations or less, unrolling further than this limit will show no effect.

**quicksort** The *quicksort* kernel sorts an array using the Quicksort algorithm.

The algorithm uses a number of loops. However, due to the complexity of the conditions of these loops, none of them are suitable for unrolling. No other possibilities for optimisation of this kernel could be found.

The array that is to be sorted during the tests with this kernel will have a size of 20.

**radixsort** The *radixsort* kernel sorts an array using the Radixsort algorithm.

The radixsort algorithm uses a number of for-loops, some with a constant number of iterations, others with a variable number of iterations. The many loops in this kernel should really be improved by applying loop unrolling. Some caching may also improve this kernel.

As with the *quicksort* kernel, the *radixsort* kernel also uses an array of size 20 for testing.

**satd** The *satd* kernel determines the sum of absolute transformed differences of a four by four field of pixels.

The kernel uses two 4-iteration loops, which can be fully unrolled to speed up the kernel. Apart from that, the kernel uses arrays extensively, so caching these array accesses may also improve the kernel.

The kernel is hard-coded to determine the sum of absolute difference of an array with 32 characters.
4.3 Results for an initial placement of the selected engines

shellsort  The *shellsort* kernel is another kernel that sorts arrays. This kernel is based on the Shellsort algorithm.

The kernel contains four for-loops, two of which are unrollable. Present in this kernel is also the use of arrays, which can be cached to improve the results further.

This kernel was tested, like the other sorting kernels, with an array of size 20.

sobel  The *sobel* kernel is part of an edge-detection program. The kernel computes the convolution of the input image with a specified mask in order to determine the gradient of the image.

The kernel uses two loops to go over each pixel of the image. These loops can be unrolled and contain loop invariant code. Inside these two loops, another two loops, each with three iterations, performs a convolution. These two loops can be fully unrolled to speed up the kernel.

The image that is used for testing this kernel is a 10-by-10 image. This puts a limit on the unrolling factor.

viterbi  The *viterbi* kernel implements an algorithm that finds a sequence of states, which are hidden from the observer. This algorithm is used in e.g. speech recognition, where the words that are uttered form the sequence of hidden states.

The kernel uses a number of for-loops. Unrolling these loops will be beneficial to most of the loops, though not for all of them.

Most of the loops used in this kernel are loops with a constant number of iterations. The specific test data will have some small influence on the execution time, but not much.

4.3 Results for an initial placement of the selected engines

This Section will use the optimisation engines selected in Chapter 3. The optimisations will be added to DWARV and tested in a number of kernels. For these tests, only the *fft*, *fir*, *floatPoint_IIR*, *FracShift*, *grid_iterate*, *loop* and *satd* kernels are used. The other kernels discussed in Section 4.2 were not yet available at the time of these tests.

4.3.1 Experimental setup

In order to evaluate the effect of each engine on the kernels, a baseline design is required to compare the results with. Each selected engine is added one by one to the baseline and any improvements made by the added engine are measured. A problem with this approach is that two of the selected engines, specifically the *demote* and the *scalarreplace* engines, are already present within the baseline. When adding, or even moving, one of these engines to DWARV, it will be difficult to measure the improvements made by these engines, since they are already present in the baseline test. Because of that, both the *demote* and the *scalarreplace* engines are removed from the baseline version of DWARV.

Since the *algebraic* engine requires that the result of an overflow in arithmetic operations is determined by taking the modulo, the appropriate option is set in the target description file. This option is discussed in more detail in Section 5.5.3. The *loopfuse*
and loopunroll engines recommend using the conevun engine before they are run. Thus, the conevpr engine was moved from after the optimisation block to its starting point and the canon option was set. Furthermore, since the scalarreplace engine operates on structures, the dismempr engine was placed at the end of the optimisation block.

To obtain the results of the cache, constprop, demote and scalarreplace engines, these engines were added one by one between the conevpr engine and the dismempr engine. To get the results of the algebraic engine, the engine was also added between the conevpr and dismempr engines, but the parallel option was set to -1 and the makemult option was set to false as well. The results of the lrrename engine were determined by adding the defuse engine, followed by the lrrename engine, in the same place where the other optimisation engines were added.

The optimisation engines that target loops all require information provided by the loopanalysis engine. Since the loopanalysis engine has been found to influence the results, a second instance of the loopanalysis engine was added with the cleanonly option, so that it will remove all information left by the loopanalysis engine. These two engines are then placed in between the conevpr and the dismempr engines. To evaluate the results of the loop optimisation engines, each engine is added between the two instances of the loopanalysis engine.

4.3.2 Results

Because of readability issues, the test results will be displayed in two figures: one figure for the optimisations that do not target loops and one figure for the optimisations that do target loops. In both figures, the baseline results refer to the setup described above, with no optimisation engine added. The results of the initial placement of the engines that do not optimise loops can be seen in Figure 4.1. The figure shows that only the cache engine shows improvements. The algebraic and the demote engines did produce changes, but the result is worse than the baseline design. All the other engines did not show any improvements nor any degradations.

The results of the loop-optimising engines, as shown in Figure 4.2, are mainly due to changes made by the loopanalysis engine. Only the loopinvariant engine shows minimal improvements of the grid_iterate kernel.

For some of the engines, these results are not surprising. Take for example the scalarreplace engine: none of the kernels made any use of structures, so there is no room for improvement from this engine to begin with. Other engines, like the loopunroll engine, did produce some surprising results. The satd kernel, for example, contains two for-loops which can be unrolled four times. Detailed output from the loopunroll and loopanalysis engines showed that the loop induction variable is not recognised. This is caused by two engines: the ssabuild engine and the ssacov engine1.

What can be learnt from these results is that there are several important factors to consider when evaluating an optimisation engine. One of these factors is the effect of all the different engines that are used in DWARV on the optimisation engine. Take for example the previously mentioned SSA engines. There is a restriction on the placement

---

1 Single Static Assignment: each time a value is assigned to a variable, a new variable is introduced instead.
4.3. RESULTS FOR AN INITIAL PLACEMENT OF THE SELECTED ENGINES 45

![Figure 4.1: Results of the initial placement of the non-loop-optimising engines](image1)

![Figure 4.2: Results of the initial placement of the loop-optimising engines](image2)
CHAPTER 4. PRELIMINARY RESULTS

of the loopunroll engine: it should not be placed after these two SSA engines. There are also dependencies on other engines. Take again the loopunroll engine: it will not work without information that is provided by the loopanalysis engine.

Another factor is the kernels that are used. When testing, for example, the scalar-replace engine with kernels that do not use any structures at all, obviously no improvements are expected. Therefore a set of kernels should be chosen that allow an optimisation engine to improve it.

4.4 Dependencies between engines

In Section 4.3 it was discovered that engines heavily influence each other. The lack of results from the tests performed in that Section is the reason the interactions between engines should be investigated further. This Section will provide a preliminary investigation of the dependencies between engines. A more complete overview of dependencies will be presented in Chapter 5.

4.4.1 Description of dependencies

The descriptions of dependencies are based upon the manual pages supplied by CoSy[5] and will focus on dependencies with other engines that are used by DWARV. The structure that is used to describe the dependencies of each engine is as follows:

1. Some engines require information that is provided by other engines. Those other engines are listed here.

2. Engines that change the IR in such a way that the optimisation engine achieves better results are listed here.

3. Engines that either limit the optimisation engine or that would benefit from being run after the optimisation engine are listed here.

algebraic

1. This engine does not need any information from other engines.

2. The cse engine may need to run before the algebraic engine. Both these engines optimise the same code, but it is very unlikely that both engines can optimise that code in a single run. Priority should be given to the engine that is expected to provide better general results.

3. The same arguments as in item 2 may hold here.

cache

1. This engine does not need any information from other engines.
2. This engine needs to determine whether or not two memory accesses refer to the same address. Therefore constant folding and constant propagation should be applied before the use of this engine.

3. There are no engines that improve the result further if run after the cache engine.

**constprop**

1. This engine does not need any information from other engines.

2. No dependencies on other engines that should be run before this engine were found.

3. Since the constprop engine introduces new constant values, the conevpr should be run after this engine. That way, a more in depth constant evaluation can take place.

**demote**  No information could be found on how the demote engine decides whether the size of an operation can be changed or not. Therefore, it is unclear how other engines will interact with the demote engine.

**loopanalysis**

1. This engine does not need any information from other engines.

2. Use of constant evaluation and constant propagation will allow this engine to determine the bounds of loops more accurately.

3. The ssa engines should be run after the loopanalysis engine. The ssa engines rewrite all variables, including the loop induction variables. The loopanalysis engine is unable to proper analyse these variables.

   Please note that when the information provided by the loopanalysis engine is no longer needed, the engine should be rerun with an option set to clean the information the engine left behind. If the information is not properly cleaned up, it can have negative impact on the execution time.

**loopfuse**

1. This engine requires the information that is provided by the loopanalysis engine.

2. In order to understand the impact of other engines on the loopfuse engine, it is important to understand the constraints that the loopfuse engine puts on two loops. First, the two loops that are candidate to fusing should be adjacent. That means that if there is any statement in between the two loops, the two loops cannot be fused. Secondly, the two loops should have the same number of iterations. Finally, the data that is used should be independent from the data processed in the current and/or future iterations of the other loop.
Considering the first constraint, any engine that can put a statement in front of, or after, a loop should not be run before the loopfuse engine. Such engines include the loophoist and the loopinvariant engines, which put loop-invariant code in front of the loop. The same holds for the loopunroll engine, since that engine can insert an entire new loop between two loops, depending on the iteration count and the unrolling factor. The loopscalar engine should not be run before the loopfuse engine as well, since that engine will add a statement after a loop.

In order to properly detect whether or not two loops have the same number of iterations, the copyprop, constprop and conexpr engines should be run before the loopfuse engine. Running those engines will make it easier to meet the second requirement.

3. Since the loopfuse engine cannot properly detect the dependencies of pointers, the dismempr engine should be run after the loopfuse engine.

**loophoist**

1. This engine requires the information that is provided by the loopanalysis engine.

2. No dependencies on other engines that should be run before this engine were found.

3. This engine applies analysis of its own. For the same reason that the loopanalysis engine has to run before the ssa engine, the ssa engines should also be run after this engine.

The loopunroll engine should also be run after the loophoist engine. Otherwise, the loopunroll engine introduces multiple copies of the loop invariant code. If the loop invariant code motion engines are run first, that will not happen, since the loop-invariant code will be removed from the loop body first.

**loopinvariant** This engine has the same dependencies as the loophoist engine, except for an additional dependency on the loophoist engine itself. The loophoist engine should be run before the loopinvariant engine. If these two engines are run the other way around, the expressions that are part of a loop invariant statement will be taken out first, after which the statement will be moved. This would result in extra variables and dependencies.

**loopscalar**

1. This engine requires the information that is provided by the loopanalysis engine.

2. The dismempr engine should be run before this engine. The loopscalar engine analyses memory accesses but cannot recognise array references and structure accesses. Therefore, these should be lowered by the dismempr engine before the loopscalar engine.
3. This engine applies analysis of its own. For the same reason that the loopanalysis engine has to run before the ssa engine, the ssa engines should also be run after this engine.

The cse engine should also come after the loopscalar engine. If an address is referred to multiple times within a loop, the cse will recognise the address as a common subexpression. As a result it is harder for the loopscalar engine to recognise the address as a target for rewriting, if possible at all.

**loopunroll**

1. This engine requires the information that is provided by the loopanalysis engine.

2. Loop unrolling works best if the number of iterations is known. Therefore, the constant propagation and evaluation engines should be run before this engine.

3. This engine applies analysis of its own. For the same reason that the loopanalysis engine has to run before the ssa engine, the ssa engines should also be run after this engine.

In the process of unrolling a loop, statements are copied and modified multiple times. This process makes it likely that the unrolled loop contains common subexpressions. Therefore, the cse engine should be run after the loop unrolling has been done.

**lrrenum**

1. This engine requires the information that is provided by the defuse engine.

2. No dependencies on other engines that should be run before this engine were found.

3. This engine acts as a special case of the ssa engines. If the ssa engines are run before this engine, this engine will not be able to make any changes, as they have already been made by the ssa engines.

**scalarreplace**

1. This engine does not need any information from other engines.

2. No dependencies on other engines that should be run before this engine were found.

3. This engine analyses and rewrites local structures. Since the dismempr engine lowers structures into memory references, no more structures will be present after the dismempr engine. Therefore, that engine should be run after the scalarreplace engine.
CHAPTER 4. PRELIMINARY RESULTS

4.4.2 Results

This Section will discuss tests that were run while taking care of the above mentioned dependencies. The setup for these tests is much the same as the setup described in Section 4.3.1 but the optimisation engines, and other added engines like e.g. the `conexpr` engine, are added between the `setlocarr` engine and the `globcse` engine. Between these engines, all mentioned dependencies are met.

Because of readability issues, the test results will again be displayed in two figures: one figure for the optimisations that do not target loops and one figure for the optimisations that do target loops. The results of the optimisations that do not work on loops are found in Figure 4.3. When compared with the results from Figure 4.1, there is an improvement. Especially the `demote` engine shows a speedup of 1.17 for the `fft` kernel. Slowdowns caused by this engine in the `satd` and `loop` kernels are no longer present in Figure 4.3. This is also the case for the `algebraic` engine and the `grid_iterate` kernel. The new placement does not provide different results for all other engines.

![Figure 4.3: Results of the placement of the non-loop-optimising engines, taking into account dependencies.](image)

Figure 4.3 shows the results of the loop optimising engines. The `loopinvariant` engine, the only engine to show any improvements in Figure 4.2, obtained an even better speedup for the `grid_iterate`. The `loopscalar` engine benefits from its new location between other engines as well. In the initial placement, no improvements were made to any of the kernels, but when taking into account the dependencies, the engine improves both the `satd` kernel and the `FracShift` kernel. The most promising engine, based on these results, is the `loopunroll` engine. It offers the most speedup for the `satd` and the `FracShift` kernels.
4.5. CONCLUSION

The loop kernel is also improved by loop unrolling, achieving a speedup of over 2.5.

![Figure 4.4: Results of the placement of the loop-optimising engines, taking into account dependencies](image)

4.5 Conclusion

The most important conclusion of this Chapter is that when placing an engine between other engines, it is important to consider the effects that engines have on each other. A good example of a kernel that illustrates this is the loop kernel. This kernel went from no speedup in Section 4.3 to a speedup of 2.7 in Section 4.4 simply by inserting the loopunroll engine into DWARV in a different location. This example shows how important it is to properly consider the effect of other engines in DWARV on the optimisation engines, but also to consider the effects of optimisations engines on each other.

This Chapter has also shown that the loopunroll, cache, loopscalar and loopinvariant engines can speedup kernels. The demote engine can both improve and worsen the execution time of a kernel and the algebraic engine causes a drop in execution time for one kernel. All other optimisation engines did not improve nor worsen the execution time of a kernel.
5.1 Introduction

This Chapter will explain how the different optimisations engines are integrated with DWARV. However, some of the engines selected in Chapter 3 and evaluated in Chapter 4 will not be discussed in this Chapter. Both the demote engine and the scalarreplace are already present in DWARV. Therefore, no further attention will be given to those engines. The constprop engine will be added to DWARV as an enabling engine, together with two other constant evaluation and propagation engines, and will not be discussed separately. The optimisations performed by the brenname engine are a subset of what the ssa engines do. Therefore, it will not be implemented as an optimisation on its own, but, if required, as an enabling engine for other optimisations. Finally, Section 4.4 has shown that the loopfuse engine has strict requirements for loops that can be fused together. These requirements are so strict that it is unlikely that loops that meet those requirements can be found in real-life applications. Therefore, the loopfuse engine is considered no further.

Before any of the optimisation space exploration techniques from Section 2.3 can be applied, all optimisations should be fully tested and integrated with DWARV. The ordering of the different optimisation engines that will be determined in this Chapter will be based upon an analysis of the dependencies between different engines. Each engine also comes with a set of options to customise its behaviour. To make the best use of the engines, each of those options are also discussed in this Chapter.

This Chapter will start by explaining how loop unrolling is integrated with DWARV in Section 5.2. Section 5.3 will show how the loop invariant code motion engines, being the loophoist and loopinvariant engines, are added to DWARV. The cache and loopscalar engines will be discussed in Section 5.4. Next, the implementation of the algebraic engine is discussed in Section 5.5. Each of these sections first explains the optimisation techniques, secondly discusses the various options that can be used to tweak the engines and finally discusses the implementation of the engine into DWARV. The Chapter will be concluded in Section 5.6.

5.2 Loop unrolling

5.2.1 Introduction

Loop unrolling is an optimisation technique commonly used to speedup loops. Instead of running the body of the loop once per iteration, loop unrolling changes the loop so that n bodies are run per iteration. Here n is a positive integer and is usually referred
to as the unrolling factor.

**Listing 5.1: Example of loop unrolling: the original loop**

```c
for ( i = 0; i < 8; i++ ) {
    array[i] = i;
}
```

Take for example the loop depicted in Listing 5.1. The Listing shows a simple loop, written in C, which assigns the index of an array element as its value. Listing 5.2 shows an unrolled version of that loop. Please note that in this example, the unrolling factor is two.

**Listing 5.2: Example of loop unrolling: the unrolled loop**

```c
for ( i = 0; i < 8; i += 2 ) {
    array[i] = i;
    array[i+1] = i+1;
}
```

The unrolled loop will require less iterations than the original loop, but every iteration twice the work should be done. One of the advantages of loop unrolling is that this technique reduces the loop overhead. The unrolled loop will require fewer comparisons between \( i \) and 8 and thus less branches as well.

Another advantage of loop unrolling is that it exposes Instruction Level Parallelism (ILP). In the unrolled loop from the previous example, the two statements in the loop body do not depend on each other. Therefore, they can be executed in parallel. For the architecture DWARV is targeting, this means that both instructions can be scheduled in the same cycle, causing further speedups. That is assuming that two writes to the memory can take place during the same cycle.

Loop unrolling works well in cases where the iteration count is known in advance. However, this is not always the case. Fortunately, loop unrolling can also be made to work with loops of an unknown iteration count by adding a pre-processing loop. This loop performs a number of iterations of the original loop while ensuring that the remaining number of iterations is a multiple of the unroll factor.

### 5.2.2 Options

Each engine as provided by ACE comes with a number of options. These options can be used to tweak the results that are produced by the engine in such a way that the results are optimal for hardware generation. This Section will discuss the options of the loopunroll engine.

- The `maxsize` option is used to influence the unrolling factor. The value of the option indicates the maximum number of nodes that an unrolled loop body is
allowed to have. This option should be used to make a decent trade-off between
the speedup of unrolling and the additional area that is required for that speedup.
The loopunroll engine should unroll as much as possible within the limits sets by
this option. Since the effect of this option on the area can only be determined after
synthesis, it is left at the default value for now.

• The maxfactor option is used to set a maximum unrolling factor. As mentioned
before, the best approach to determine the unrolling factor is to use the maxsize
option to limit the number of loop bodies. Therefore, the maxfactor option should
be sufficiently high, e.g. 128.

• The loosen option allows the engine to relax the conditions a loop should meet to
be unrolled. This option is present for testing purposes and should be disabled.

• The fullunroll option can be used to allow unrolling if and only if a loop can
be unrolled fully. Considers for example the fft kernel used in scientific applica-
tions. Such kernels contain loops spanning many iterations. For this kind of loops,
unrolling them fully is simply not possible, due to area constraints. However, by
unrolling such a loop only partially, a good speedup is achieved. Thus, unrolling
should not be limited to loops that can be fully unrolled and this option should be
set to false.

• The loopfactor option allows loops for which an unrolling factor cannot be de-
termined to be unrolled with the specified factor. As is the case with the loosen
option, this option is present for testing purposes and should be disabled.

• The noreplace option is used to prevent the replacement of the loop-induction
variable. If it is set to true, the update statement of the induction variable, e.g.
\( i++ \), will not be replaced by e.g. \( i+ = 4 \), in the case where the unroll factor is four.
If the updating statement is replaced, instead of e.g. four updating statements,
there is only a single statement. This is more efficient due to the fact that there
are less dependencies to take into account. Thus this option should be set to false.

• The noreplaceall option allows the engine to only replace the loop-iteration
counter, but not the other loop-induction variables. As mentioned with the
noreplace option, the more variables are replaced, the more efficient the result
will be. Therefore, this option should be set to false.

• The levels option is used to limit the number of nested loops that will be unrolled.
To extract as much parallelism as possible, as much loops as possible should be
unrolled. Hence the value of this option should be infinity. But since this option
cannot be set to infinity, it should be set to a value that is sufficiently high, for
example 128.

• The hwloops option allows the engine to unroll so-called ‘hardware loops’. These
‘hardware loops’ are special loops that are used by a software pipeliner. Since these
hardware loops are currently not in use by DWARV, this option can best be set to
false. Should software pipelining be implemented in the future, this option should
be set to true in order to maximise the parallelism of the loops that are pipelined.
• The `whiledoloops` option allows the unrolling of a loop with a while-do structure with an unknown number of iterations. Restricting the engine by not unrolling such a loop would severely reduce the engine’s potential for speedup, since most loops are translated into either a while-do loop or a do-while loop. Thus, this option should be set to true.

• The `repeatuntilloops` option allows the unrolling of a loop with a do-while structure with an unknown number of iterations. For much the same reason as the `whiledoloops` option, this option should be set to true.

• The `constant` option is used to unroll only loops that have a constant number of iterations. Even though for loops without a constant number of iterations require guarding statements and statements to make the loop iterate a multiple of the unrolling factor times, the speedup gained by applying `loopunroll` is expected to be larger than the loss caused by this overhead. Therefore, limiting the unrolling to loops with a constant iteration count should not be done and the option should be set to false.

• The `factors` option limits the possible unrolling factors. A loop will only be unrolled with an unrolling factor that is in the list specified by this option. The default value is to allow all possible unrolling factors. Without any reason to limit the unrolling factor, this option should be left to its default value.

A short summary of the recommended values for each option can be found in Table 5.1.

Table 5.1: Default and recommended option values for the `loopunroll` engine.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maxsize</code></td>
<td>1024</td>
<td>? (leave at 1024 for now)</td>
</tr>
<tr>
<td><code>maxfactor</code></td>
<td>4</td>
<td>128</td>
</tr>
<tr>
<td><code>loosen</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>fullunroll</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>loopfactor</code></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><code>noreplace</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>noreplaceall</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>levels</code></td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td><code>hwloops</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>whiledoloops</code></td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td><code>repeatuntilloops</code></td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td><code>constant</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>factors</code></td>
<td>”1:”</td>
<td>”1:”</td>
</tr>
</tbody>
</table>
5.2. LOOP UNROLLING

5.2.3 Implementation

In Section 4.4 it was discussed in what way the selected optimisation engines interact with each other and with the engines already present within DWARV.

As mentioned, the loopunroll engine requires the loopanalysis engine to have finished and the ssa engines should not have been run yet. An overview of the dependencies between engines can be seen in Figure 5.1.

![Diagram of dependencies between engines]

No dependencies were mentioned between the constant propagation and evaluation engines and the loopanalysis engine. Nevertheless, those engines can change the use of variables that are used by the loopanalysis engine. Thus, another dependency arises: the constant propagation and evaluation engines need to be run prior to the loopanalysis engine.

Of the engines that are needed, the copyprop, conepr, cse (as globcse) and ssa engines are already present in DWARV. The copyprop and conepr engines are placed after the ssa engines. The globcse engine is run just prior to the ssa engines.

To allow for a fair comparison with the original version of DWARV, the choice was made to insert the loop unrolling engines just before the globcse engine. The copyprop and conepr engines were duplicated. One copy remained in the original location and the other copy was placed before the loopanalysis engine.

The loopanalysis engine should ideally be run as close to the loopunroll engine as possible. The more engines that are run in between, the higher the chance that the analysed data is incorrect.
While none of the added engine is dependent on the \textit{looppragma} engine, this engine was added to DWARV. Placed directly after the \textit{loopanalysis} engine, this engine allows more advanced users of DWARV to pass more detailed information to the \textit{loopunroll} engine by using \texttt{#pragma loop <directive>} inside the source code. Possible directives are listed in Table 5.2.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxintercount(int)</td>
<td>The maximum number of times a loop is known to iterate.</td>
</tr>
<tr>
<td>minintercount(int)</td>
<td>The minimum number of times a loop is known to iterate.</td>
</tr>
<tr>
<td>no unroll</td>
<td>Prevent the \textit{loopunroll} engine from unrolling the loop.</td>
</tr>
<tr>
<td>unroll(int)</td>
<td>The number of times to unroll a loop.</td>
</tr>
<tr>
<td>maxunroll(int)</td>
<td>Provides an upper limit to the unrolling factor.</td>
</tr>
</tbody>
</table>

5.2.3.1 Problems encountered

During the implementation and testing process of the \textit{loopunroll}, a number of problems were encountered. Those problems, and in most cases also their solutions, are presented in the remainder of this Section.

Adding support for logical AND statements In Section 5.2 it was mentioned that a pre-processing loop is necessary when a loop has an unknown number of iterations. This loop will need to verify that the loop should run at all and it should ensure that the number of iterations after this loop is a multiple of the unrolling factor. For this, DWARV will need to be able to use a logical AND statement.

DWARV handles conditional statements, primarily if-statements, by using a dedicated signal: the Condition Code (CC) signal. When code for a conditional statement is generated, first a value of 1 or 0 should be assigned to CC. Suppose this assignment happens in cycle \(i\). During cycle \(i + 1\), the state-machine will determine, depending on the value of the CC signal, which state will be executed in cycle \(i + 2\).

Originally, DWARV supported multiple conditions by splitting the conditions over several different if-statements during the lowering process. For generating code for multiple conditions after this lowering is done, changes to the method that handles conditional statements are required. Either more CC signals, that can be combined, are necessary or the conditions should be written into a regular register and the CC signal should be read from that register. Resource-wise both solutions are close to each other. There is no fundamental difference between a ‘regular’ register and a condition register. However, the generation of the state machine will be more complicated with the former approach. This is the case because the state machine will need to know which CC signal to check. With the latter approach, there is only a single CC signal, thus simplifying the state machine. For that reason, the second approach to solving this problem will be used. This scheme, with only a single CC signal is not optimal since a single signal will be used.
5.2. LOOP UNROLLING

throughout the design, but a rewrite of the way DWARV handles conditional statements is not the topic of this thesis.

Using the second scheme means that a logical AND should be able to write to the CC signal. Existing rules for bitwise ANDs take two registers and will store the result in another register. Using these rules as a basis, the rule from Listing 5.3 was created.

Listing 5.3: The and_int_cc rule

```vhdl
1 RULE [and_int_cc] o:mirAnd(rs1:reg, rs2:reg) -> cc:cc;
2 CONDITION { IS_BOOL(o.Type) };
3 EXPAND emit {
4 printf (OUTFILE,"tCC <= %s AND %s:\n", REGNAME(rs1),
5 REGNAME(rs2));
6 }
```

This rule will match a mirAnd, which has two registers as parameters, and it will return a condition code. Additionally, this rule should only be used if the mirAnd is of the boolean type. If such an AND is found in the IR, the EXPAND emit clause will be executed. For this rule, a VHDL signal assignment statement will be printed in the output file.

Now that a mirAnd can write to the CC signal, conditional expressions should be capable of writing to registers. Without support for this, a mirAnd would not be able to read these conditions. Apart from the newly created rule for and-expressions, only comparisons write to the CC signal. In fact, the loop unrolling engine only inserts a logical AND consisting of two comparisons between registers. As a side note, the front-end will generate nested if-statements in the form as shown in Listing 5.4.

Listing 5.4: CoSy support scheme for a logical AND

```vhdl
1 if (cond1) then
2   if (cond2) then
3     do_work()
4   endif
5 endif
```

As mentioned before, the loopunroll engine will only require support for comparisons between two registers. Code for comparisons, indicated by a mirCompare node, is generated in two phases. The first phase is a wrapper that will extract the type of condition from the mirCompare and calls an instruction to generate VHDL. This instruction will then check the type of condition and output VHDL that calls a corresponding VHDL function. The result of this function is assigned to the CC signal.

In Listing 5.5 the wrapper is modified for writing the result of the comparison to a register. The modifications that were made are the instruction that is called
Listing 5.5: Wrapper for the mirCompare code-generation

```vhdl
RULE [cmp_reg_to_reg] c:mirCompare(s1:reg, s2:reg) -> rd:reg;

CONDITION{
  IS_INT(mirEXPR_get_Type(mirEXPR_mirCompare_get_Left(c))) &&
  IS_INT(mirEXPR_get_Type(mirEXPR_mirCompare_get_Right(c)))
};

EXPAND{
  Condition cc;
  cc = rel_to_cond(c.Relation, mirEXPR_get_Type(c.Left));
  gcg_create_cmp_to_reg(gcg_expand, s1, s2, cc, rd);
};

END
```

The instruction that is called, see Listing 5.6, will check what kind of condition is used to compare two registers. Depending on the condition, the instruction will generate VHDL code that calls an appropriate function to do the comparison. These functions are defined earlier in the VHDL output. Instead of writing the result of the comparison to the condition code, as is done by the original instruction, this instruction will put its return value into a register.

Finally, DWARV will need to know about the size of the registers that will contain conditions. Obviously, a condition requires only two distinct values: true or false. Therefore, a register width of one bit will suffice. DWARV uses an engine called sizecalc to determine the size of an expression, based on the CCMIR type. The size_of_type function inside this engine uses a switch-statement to check what the CCMIR type of an object is. A new case for the mirBoolean type has to be added, where the case specifies a bitwidth of one.

Adding all these features together will allow DWARV to generate code for conditional statements that make use of a logical AND.

**Adding the rewrite engine** During the testing of the *loopunroll* engine, another problem was encountered. It appeared that DWARV could not generate VHDL for an if-statement with a constant value of false as condition. This problem was not caused by the *loopunroll* engine, but by the *constprop* engine. A variable inside the *fft* kernel was assigned a constant value of false. The *constprop* took this constant value and put it inside the if-statement. Since the if-statement was originally only checking the variable, this created an if-statement with a constant condition.

This problem was not hard to fix, since rewrite rules to deal with such if-statements were already present. The *cg* engine did not call these rules. Adding the *rewrite* engine just prior to the *match* engine solved this problem.
5.2. LOOP UNROLLING

Listing 5.6: Code-generating instruction for mirCompare

1  INSTRUCTION cmp.to_reg (rs1:reg, rs2:reg) \rightarrow rd:reg;
2  ATTRIBUTE cond:Condition;
3  EXPAND emit {
4
5      switch(cond) {
6          case cc_equ:
7              printf (OUTFILE, "t%s <= equal ( %s , %s );\n" ,
                      REGNAME(rd), REGNAME(rs1), REGNAME(rs2));
8              break;
9          case cc_neq:
10              printf (OUTFILE, "t%s <= not_equal ( %s , %s ) ;\n" ;
                      REGNAME(rd), REGNAME(rs1), REGNAME(rs2));
11              break;
12          ...
13      };
14  END

Fixing logical AND expressions The shellsort kernel compiled without any problem. However, the simulation results were incorrect. These inconsistencies were caused because the pre-processing loop did not execute correctly. A more detailed analysis of the simulation made it clear that the CCU jumped a cycle too soon in the pre-processing loops.

With the source of the problem identified, a more detailed analysis of the problem was possible. The condition of the pre-processing loop utilised the logical AND that was implemented in Section 5.2.3.1. The condition code was both assigned and read in the same cycle. This caused the jumps to use the previous value of the CC signal, thus sometimes jumping when it should not (and vice versa). This happens because DWARV does not know about the dependency between the logical AND, which assigns a value to the CC signal, and the jump that uses the CC value. This is due to an omission in the code found in Listing 5.3. Adding an extra line containing WRITE REGISTER <CC>; will fix this problem. See Listing 5.7 for the final rule.

dom failure with UNIQINT While compiling the boyermoore kernel with loop unrolling enabled, another error occurred. An “Internal fatal error in engine rex”. Most if statements will be followed by either the ‘then’ basic block or the ‘else’ basic block, both of these are handled by the if someisnext rule. However, when this is not the case, a different rule has to be used, the if rule. What happens with the boyermoore kernel is that such an if-statement is generated by the loopunroll engine. The rule then attempts to create a new basic block with a unique label. However, this last step is failing. The internal error occurs during a call to the UNIQINT_NextLabel function. Since this function is provided by the CoSy framework, it is not possible to provide a fix for this
Listing 5.7: The final and_int_cc rule

```c
RULE [and_int_cc] o:mirAnd(rs1:reg, rs2:reg) -> cc:cc;
CONDITION { IS_BOOL(o.Type) };
WRITE REGISTER <CC>;
EXPAND emit {
    printf (OUTFILE, "tCC <= %s AND %s;\n", REGNAME(rs1),
               REGNAME(rs2));
};
END
```

Adding support for non-integer types  During the testing of the loopunroll engine, it was discovered that the engine had problems with a loop in the hamming kernel. The kernel contains a 16-iteration loop, which should be unrolled. However, this did not happen, because the iteration counter was defined as a short integer. The C standard specifies that arithmetic done with types, which are smaller than an integer, should be done with integers. This is called integer promotion. As a consequence, type conversions from short integer to integer are added to the IR. This means that the analysis engine sees these type conversion, instead of a variable. The demote engine removes these conversions. After placing the demote engine before the loopanalysis engine, the loop with the short integers was indeed unrolled.

5.3 Loop invariant code motion

5.3.1 Introduction

Loops can contain any kind of expressions and statements. One special type of code is the so-called loop invariant code. This type includes expressions and statements that are placed inside a loop, but is actually constant for all iterations. When inside a loop, the code is executed every iteration. But if this code is placed before the loop, it will only be executed once and the loop will execute faster. The loop body contains less expressions and statements and any code dependent on the loop invariant code will no longer have to wait before the results of the loop invariant code are known. This technique to optimise a design is known as loop invariant code motion.

Within the CoSy framework, there are two engines that apply loop invariant code motion: the loophoist and loopinvariant engines. The loophoist engine performs the code motion on entire statements while the loopinvariant engine applies loop invariant code motion on expressions. The implementation of these two engines will be discussed in this Section.
5.3. LOOP INVARIANT CODE MOTION

5.3.2 Options

Each engine as provided by ACE comes with a number of options. These options can be used tweak the results that are produced by the engine in such a way that the results are optimal for hardware generation. This Section will discuss the options of the loophoist and loopinvariant engines.

5.3.2.1 Options for the loophoist engine

The loophoist engine offers four options to modify its behaviour: the guard option, the levels option, the indvars option and the loose.real option.

- The guard option is used to determine whether or not a guarding statement surrounds the loop-invariant code. The guarding statement is necessary to determine whether or not a loop, and by inference the loop-invariant code, should be executed at all. To ensure that the output of DWARV with or without running this engine will be the same, this option should be set to true.

- The levels option controls how deep a loop can be nested if loop hoisting is applied. For hardware generation, as much invariant code as possible should be moved and there should not be a limit to the nesting level of a loop. Therefore, this option should be set to minus one, which means that there is no limit to the nesting level.

- The indvars option attempts to remove induction variables within a loop that has a fixed number of iterations. Induction variables will be removed if they are invariant within a loop. The removal of unnecessary variables will reduce the area that has to be used and the number of possible dependencies between variables. Therefore, this option should be set to true.

- The optimisation mentioned under the indvars option is not applied on floating point numbers. The reason for this is that the optimisation will cause a change in precision. By setting the loose.real option to true, the optimisation will also be applied to floating point numbers. Until the effect of the difference in precision is investigated, this option should be set to false.

A short summary of the recommended values for each option can be found in Table 5.3.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td>guard</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>levels</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>indvars</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>loose_real</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
5.3.2.2 Options for the loopinvariant engine

In order to properly discuss the options of the loopinvariant engine, the concept of an ‘expensive expression’ should be explained first. The engine will only move these ‘expensive’ expressions outside of a loop. Any expressions that are not ‘expensive’, even though it might be loop-invariant, will not be moved outside the loop. Which expressions are considered ‘expensive’ by the engine can be configured using the options of the engine.

- The levels option is similar to levels option of the loophoist engine from Section 5.3.2.1 and should be set to minus one for the same reasons.

- The globals option can be used to ensure that the loopinvariant engine considers a global address expensive. Since DWARV does not support global variables (including addresses) this option should be set to false.

- The fpconst option allows for floating point constants to be considered expensive operations. VHDL operations, e.g. an and, which use constants can be synthesised very effectively. A specialised block can be created for the constant operation. On the other hand, if these constants are moved from where they are used, additional signals, and thus registers and wires, are required. Therefore, it is more efficient to leave the constants where they are used. Thus this option should be set to false.

- The fixconst option allows for fixed point constants to be considered expensive. For the same reason as the fpconst option, this option should be set to false.

- The limit option is used to determine which normal expressions are expensive. Any expression that contain a number of subexpressions equal or greater than set by this option will be considered expensive. Until this option is looked at in more depth, the default value of 4 is a reasonable value. If this value to too high, less expressions can be moved and thus less can be gained by using this engine. On the other hand, if the value is too low expressions that are trivially small will be moved. This will cause extra overhead without causing speedup. On top of that, the synthesis of the kernel will become more difficult, due to more and longer wiring.

- By setting the locarr option to true, references to local array elements will be considered expensive. The idea behind this option is that on a ‘normal’ architecture local arrays are put into the memory. That would make references to a local array expensive. But since a local array in DWARV consists of signals, just as normal variables, they should not be treated specially. Therefore, this option should be set to false.

- The op1 option allows unary operators to be specified. These operators will be considered as being expensive. At the moment there is no reason to treat any operation specially. Therefore, this option should be left at its default value.

- The op2 option operates the same as the op1 option, except for the fact that it operates on binary operators instead of unary operators. As with the op1 option, the value of this option should be left at the default value.
• The `ptrcvt` option will allow the engine to consider the conversion of a pointer as if it were no operation. This option should be used when a conversion between pointers should not count when determining if an expression is expensive. DWARV does contain rules that will generate VHDL for pointer conversions. Therefore, this conversion should also be considered, since it does use up ‘time’. This option should be false.

• The `aggr_types` option enables the more complex types like structures and arrays to be considered expensive. However, since these types are already lowered before the `loopinvariant` engine is called, enabling these types will make no difference. Therefore, this option should be kept at its default value.

• The `offset_lo` option sets a lower limit for offset used with pointer addressing. The same option is used in the `globcse` engine with a value of zero.

• The `offset_hi` option sets an upper limit for the offset of pointer addressing. The same option is used in the `globcse` engine with a value of 4,294,967,295. With this value, 32-bit unsigned offset is possible.

• The `offset_scale` option allows the engine to consider the size of an element for determining an offset. This option is not used by the `globcse` engine and should not be used here.

• The `callsafe` option allows optimisations even with function calls inside a loop. Since DWARV inlines all function calls, this option is not relevant and should be kept at its default value.

• The `safecalls` option specifies functions that are considered to be safe. If the `callsafe` option is false, any function indicated by this option is considered safe for processing. Since function calls are inlined by DWARV, this option is not relevant to DWARV.

• The `cost_hi` option is used to determine whether or not an expression is expensive based on predefined costs. Costs can be predefined using a separate engine. This option specifies a lower limit. If the cost of an expression is higher than this value, the expression is expensive. Without an engine that can adequately determine the costs of expressions, the value of this option should be left at its default value.

• The `cost_lo` option is used to determine whether or not an expression is expensive based on predefined costs. This option specifies an upper limit. If the cost of an expression is lower than this value, the expression is never expensive. Without an engine that can adequately determine the costs of expressions, the value of this option should be left at its default value.

• Using the `init_in_preehdr` option will place the moved code into the pre-header part of a loop instead of in the initialisation part. This would mean addition basic blocks will be required. Hence, to keep the control-flow as simple as possible, it is best to set this option to false.
A short summary of the recommended values for each option can be found in Table 5.4.

Table 5.4: Default and recommended option values for the loopinvariant engine.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td>levels</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>globals</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>fpconst</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>fixconst</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>limit</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>locarr</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>op1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>op2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ptrcvt</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>agrgr_types</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>offset_lo</td>
<td>$-4 \ 096$</td>
<td>0</td>
</tr>
<tr>
<td>offset_hi</td>
<td>$4 \ 095$</td>
<td>$4 \ 294 \ 967 \ 295$</td>
</tr>
<tr>
<td>offset_scale</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>callsafe</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>safecalls</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>cost_hi</td>
<td>9 999</td>
<td>9 999</td>
</tr>
<tr>
<td>cost_lo</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>init_in_prehdr</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

5.3.3 Implementation

In Section 4.4 it was discussed in what way the selected optimisation engines interact with each other and with the engines already present within DWARV.

As mentioned, the loophoist engine should run before the loopinvariant engine and both these engines should be run before the ssa engines and before the loopunroll engine. On top of that, the loopinvariant engine would benefit from the results of the conevpr engine. In Section 5.2.3 it was mentioned that the conevpr engine should be run before the loopanalysis engine.

Since neither of the added engines is reported to invalidate the information provided by the loopanalysis engine, it is assumed that the loop-analysis should not be redone after this engine. Therefore, no additional loopanalysis engines are necessary.

Figure 5.2 shows these dependencies in a graphical way.
5.4 Software-sided caching

5.4.1 Introduction

This Section will discuss an optimisation technique, based on the notion of caches. A traditional cache is a low-latency memory device that operates in between the CPU and the main memory. The use of a cache reduces the average latency of memory operations. A similar effect can be achieved by rewriting software. Suppose for example that a program writes a value to address $x$ in the memory. A number of instructions later, the program reads address $x$ from the memory. Such a program would require at least two memory operations: one write and one read. Now suppose that the program is rewritten. Instead of writing to memory location $x$, the value of the write is assigned to a temporary variable $t$. When $x$ is read again, instead of accessing the memory, the value can be read from variable $t$. Finally, $t$ will be written to the memory after the last write to memory location $x$. This will need only a single memory operation.

This effect can be achieved within DWARV using the cache and loopscalar engines. The loopscalar engine applies this caching on memory references within a loop, while the cache engine works on all other memory references. The integration of these two engines with DWARV will be discussed in this Section.
CHAPTER 5. INTEGRATION OF SELECTED OPTIMISATION ENGINES WITH DWARV

5.4.2 Options

Each engine as provided by ACE comes with a number of options. These options can be used to tweak the results that are produced by the engine in such a way that the results are optimal for hardware generation. This Section will discuss the options of the loopscalar and cache engines.

5.4.2.1 Options for the loopscalar engine

The loopscalar engine also uses the notion of ‘expensive’ expressions as introduced in Section 5.3.2.2. Expensive expressions are candidates which can be cached.

- The globals option is similar to the globals option from the loopinvariant engine. As discussed in Section 5.3.2.2, the value should be kept at its default value.

- The levels option is similar to the levels option from the loophoist engine. As discussed in Section 5.3.2.1, the value of this option should be -1.

- If there are any nested loops, the loopscalar engine will start analysing the outer loop first. With the inner_first option, this behaviour can be changed. Whether or not the inner loop is handled first, does not matter from the perspective of the memory as long as the same memory accesses are cached. From a control-flow point of view, it is better to run the inner loop first. The memory access will be placed in the outer loop and when the outer loop is processed, it can be placed outside the outer loop. However, the same result is possible when the loop-invariant code motion engines are run afterwards. If the loopscalar engine is run on its own, this option might be enabled, but when run before the loop-invariant code motion engines, it can remain disabled.

- The null_safe option determines how the engine deals with pointers. Pointers which might be NULL-pointers will not be cached. When using this option, it is assumed that no pointer will be a NULL-pointer. In order to ensure that no NULL pointer is accidentally referenced, this option should be set to false.

- The guard option is similar to the guard option from the loophoist engine. As discussed in Section 5.3.2.1, this option should be set to true.

- Using the read_only option, memory accesses that are read, but not written will also be cached. However, the same effect is achieved with loop-invariant code motion. Therefore, when this engine is used without loop-invariant code motion, this option should be set to true, otherwise it should be set to false.

- The write_only option is similar to the read_only option, except for the fact that it caches accesses which are not read, but only written. Using this option would mean that all writes are processed after the end of the loop. Using this option will reduce the load on the memory during the execution of the loop and should be set to true.
• If the `sure_write` option is enabled, memory accesses will be cached only if the
loop is known to run at least once. Since it cannot always be guaranteed through
the IR that a loop will run at least once, but it is highly likely that a loop will
run at least once, this option should be set to false. Assuming a kernel is well
programmed, a loop will not run if one of the inputs causes it to do so. However,
in general the loop will run most of the time.

• The `aggr_types` option is similar to the option with the same name from the
`loopinvariant` engine. As discussed in Section 5.3.2.2, the value should be kept at
its default value.

A short summary of the recommended values for each option can be found in Ta-
ble 5.5.

Table 5.5: Default and recommended option values for the `loopscalar` engine.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>globals</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>levels</code></td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td><code>inner_first</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>null_safe</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>guard</code></td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td><code>read_only</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>write_only</code></td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td><code>sure_write</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td><code>aggr_types</code></td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

5.4.2.2 Options for the `cache` engine

The `cache` engine has only a single option: the `make_extract` option. This option will
translate a load of a cached value into a smaller data type into the extraction of the bits,
using bitfield operations. The alternative is a normal type conversion. A type conversion
is safer in DWARV, since it already knows how to deal with the signedness of numbers.
Therefore, this option should be set to false.

A short summary of the recommended values for each option can be found in Ta-
ble 5.6.

Table 5.6: Default and recommended option values for the `cache` engine.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>make_extract</code></td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
5.4.3 Implementation

Some dependencies on the loopscalar and cache engines can be found in Section 4.4. The loopscalar engine, being a loop-based engine, requires the information that is provided by the loopanalysis engine. Furthermore, this engine will make changes that will leave the data from the analysis invalid. Therefore, if any loop-based engine is run after this engine, the loopanalysis engine should be run again. Since the loopscalar engine also applies its own analysis, the ssa engines should not have run yet.

Applying the caching on loops before doing loop-unrolling is preferred. Though the end result will be turn out to be the same, doing the caching after the unrolling will be more complicated and time consuming. Since both engines operate on memory accesses, constant propagation and folding will help to determine whether or not two memory accesses refer to the same address. For the best efficiency, any references to structures should be lowered by the dismempr engine before the two caching engines. These engines should be executed before the cse engine, since that engine can potentially eliminate common subexpressions in memory locations (such as e.g. the base address). Common subexpression elimination could change the addresses in such a way that they are no longer clearly recognisable.

A summary of these dependencies can be found in Figure 5.3.

Figure 5.3: Overview of dependencies concerning the loopscalar and cache engines. Green indicates the two engines that are to be added. Red indicates a required dependency. Blue indicates engines that would improve the result.
5.5 Algebraic simplifications

5.5.1 Introduction

The algebraic engine uses several known algebraic simplifications to rewrite expressions. The engine can also balance large expressions. The VHDL that will be generated for a large expression will require fewer cycles to compute the entire expression. The integration of the algebraic engine is discussed in this Section.

5.5.2 Options

The algebraic comes with two options.

- With the makemult option set to false, any simplification that will requires additional multiplications, e.g. optimising $a + a$ into $2a$, will not be allowed. At first glance, using this option to prevent unnecessary expensive multiplications will provide better results. However, all multiplications that fit into the category targeted by this option will turn out to be multiplications with a constant. Since this kind of constants are handled as special cases by DWARV and the synthesiser, these additional multiplications are not expensive. Thus, the question of whether or not to allow these additional constant multiplications boils down to the following question: is it more efficient to perform an addition or a constant multiplication? A constant multiplication can be replaced by additions in combination with bitshifts. See Chapter 9.5 in [20]. While the bitshifts are not a problem, multiple adders might be used in constant multiplication. Using a single adder is more efficient area-wise than a combination of multiple adders, so this option should be set to false.

- The parallel option controls to what degree parallelism can be recovered in chains of expressions. This can range from a left-recursive expression to a fully balanced expression. The more left-recursive an expression is, the more dependencies there will be between variables. Such an expression has a worst-case timing of $O(n)$. On the other hand, a fully balanced tree will have fewer dependencies and a worst-case timing of $O(\log n)$. Therefore, the parallel option should be set to produce a fully balanced expression, which is indicated by a negative value.

A short summary of the recommended values for each option can be found in Table 5.7.

Table 5.7: Default and recommended option values for the algebraic engine.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>makemult</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
5.5.3 Implementation

Other than a potential conflict with the cse engine, as mentioned in Section 4.4, the algebraic engine does not appear to have a dependency on other engines.

Consider for example the pseudo-code in Listing 5.8. This can be optimised by the algebraic engine and the cse engine, but not by both. If algebraic engine factorises $a$ out of the first statement and $b$ out of the second, the result will no longer have any common subexpressions. Likewise, the cse engine can eliminate the $a \cdot b$ from both statements, but no opportunities for factorisation will remain. But which engine should be given the priority?

Listing 5.8: Sample code

```
1 x = ab + ac;
2 y = ab + bc;
```

To determine the most effective engine, the number of expensive operations should be compared. In this simple example, the most expensive operations are the multiplications. The original code contains four multiplications. The code generated by the algebraic engine, see Listing 5.9, will contain only two multiplications. The code in Listing 5.10, generated by the cse engine, contains three multiplications. For such a simple code, the algebraic engine will generate the design with the least multiplications. This is also true for more complex expressions, such as e.g. polynomial expressions[11].

Listing 5.9: Sample code, optimised by the algebraic engine

```
1 x = a * ( b + c );
2 y = b * ( a + c );
```

Listing 5.10: Sample code, optimised by the cse engine

```
1 t = ab;
2 x = t + ac;
3 y = t + bc;
```

The algebraic engine is also able to parallelise large expressions. As such, the engine would be best place after engines that may create large expressions. Most optimisation engines attempt to simplify and move large expressions. The only optimisation engine which can introduce large expressions is the loopunroll engine. When two or more loop bodies become the new body of the loop, the expressions in that body can be longer and more complex. This could make them a target for the algebraic engine.

For the algebraic engine to work properly, an entry should be added to the file that describes the target machine: the Target Description (or TarDes) file. Inside the TarDes file, the OVINT options should be set to 1. The OVINT option indicates the behaviour...
5.6. CONCLUSION

of the target when an overflow occurs inside the integer unit. Setting this option to 1 means that the modulo of the result will be returned, instead of the highest possible value (saturation). Most common arithmetic units do indeed provide the module of the result in case of an overflow\cite{20}, so it is likely that the synthesiser will indeed generate arithmetic units with modulo overflow handling.

A summary of these dependencies can be found in Figure 5.4.

Figure 5.4: Overview of dependencies concerning the algebraic engine. Green indicates the engine that is to be added. Blue indicates engines that would improve the result.

5.6 Conclusion

This Chapter discussed how six out of the eleven selected optimisation engines can be integrated with DWARV. The other five optimisations were either already present within DWARV or, on second thought, did not have sufficient promise of speedup.

The six optimisation engines that were discussed in this Chapter are evaluated in Chapter 6.
6.1 Introduction

The following Chapter will discuss the results obtained from testing the optimisation engines that were discussed in Chapter 5. For these results, all kernels from Section 4.2 were used, with the exception of the loop kernel. The reason to exclude the loop kernel is that it is not a representative kernel, due to its extreme simplicity.

While all those kernels are used in every test, not all of them will be displayed in the Figures in this Chapter. If all kernels are shown in each Figure, the Figures would not be readable. Unless stated otherwise, only kernels with speedups between 0.9 and 1.1 will be excluded from the Figures.

This Chapter will start with an evaluation of the loopunroll engine in Section 6.2, followed by a discussion on the results of loop invariant code motion in Section 6.3. Section 6.4 will analyse the results of the software-sided caching technique. Section 6.5 discusses the algebraic simplifications. The effect of utilising different orderings for the engines is investigated in Section 6.6. The results obtained with these new optimisations in compared with LegUp, a similar HLS tool, in Section 6.7. Finally, this Chapter will close with a conclusion in Section 6.8.

6.2 Loop-unrolling

6.2.1 Introduction

Loop unrolling is a technique that changes a loop in such a way that it will execute multiple bodies of the original loop in a single iteration. As mentioned in Section 5.2, loop unrolling reduces the number of branches necessary for a loop and it also exposes Instruction Level Parallelism (ILP).

This Section will start with an evaluation of the loopunroll engine in Section 6.2.2. Section 6.2.3 will close this Section with a discussion on the use of the maxfactor option that was introduced in Section 5.2.

6.2.2 Evaluation of improvements

In order to assess the effectiveness of the loopunroll engine, the results of a DWARV version with and without the engine should be compared. This comparison is done in Figure 6.1. The Figure shows the results of a run with the baseline DWARV, that is DWARV without any optimisations, and DWARV with loop-unrolling enabled.

Five of the kernels, the fir, FracShift, hamming, sobel and viterbi kernels, show a good speedup of over 1.5, some even approaching the three times speedup. Three kernels show a decent speedup between 1.0 and 1.5.
There are four kernels which cause a significant drop in performance. These kernels are the bellmanford kernel, the count_alive kernel, the floydwarshall kernel and the multiply kernel. The average speedup of the loopunroll over all kernels is 1.26.

Of particular interest are the four kernels where the execution time was decreased. More specifically, the reason why the performance decreased. In order to determine this reason, the kernels should be analysed more carefully.

Listing 6.1: The loop of the count_alive kernel

```c
for ( x = i - 1; x <= i + 1; x++ ) {
    for ( y = j - 1; y <= j + 1; y++ ) {
        if ( x == i ) && ( y == j ) continue;
        if ( y < size ) && ( x < size ) && ( x >= 0 ) && ( y >= 0 ) {
            a += CELL( x, y );
        }
    }
}
```

The count_alive kernel is part of a Game of Life implementation. The kernel is responsible for the counting of the number of neighbours which are alive. This is achieved by looping through three rows, starting from the row above the cell, and three columns,
starting from the column to the left of the cell. However, the kernel is written in an unusual manner. Instead of looping from \(-1\) to \(1\), the kernel loops from \(i - 1\) to \(i + 1\), where \(i\) is one of the coordinates of the centre cell. This loop is shown in Listing 6.1.

Because of this, the loopunroll engine cannot determine these fixed bounds of three by three. A modified version of the kernel, the count_alive_mod kernel, loops from \(-1\) to \(1\) and uses additional variables to determine the coordinate of the cell that is to be checked. The modified loop is shown in Listing 6.2. According to the results from Figure 6.1, the count_alive_mod kernel achieves a speedup of 1.25.

Listing 6.2: The modified loop of the count_alive kernel

```c
for ( k = -1; k <= 1; k++ ) {
    x = i + k;
    for ( l = -1; l <= 1; l++ ) {
        y = j + l;
        ...
    }
}
```

The problem with the bellmanford and floydwarshall kernels is the structure of the loop bodies. The bodies of these kernels are governed by if-statements. Because of this conditional statement, statements from the \(i\)th loop iteration cannot be merged into the same basic block as statements from iteration \(i - 1\) or \(i + 1\) after performing loop-unrolling. Thus no additional ILP is exposed. For illustration, the main loop of the bellmanford kernel is shown in Listing 6.3.

Listing 6.3: The main loop of the bellmanford kernel

```c
for ( i=0; i < nodecount; ++i ) {
    for ( j=0; j < edgecount; ++j ) {
        if ( distance[source[j]] != INFINITY) {
            int new_distance = distance[source[j]] + weight[j];
            if (new_distance < distance[dest[j]])
                distance[dest[j]] = new_distance;
        }
    }
}
```

However, this structure is not everything that causes the slowdown. The count_alive_mod kernel has a similar structure, but does actually experience a speedup. The major difference between the count_alive_mod kernel and the other kernels is the fact that the count_alive_mod kernel contains a constant number of iterations. As mentioned in Section 5.2, when a loop has a non-constant number of iterations, a pre-processing
loop is required. This pre-processing loop in combination with no exposed ILP results in the drop in execution time.

Something similar happens with the `multiply` kernel. The statements in the main loop of this kernel are dependent on the results of previous iteration of the loop. Due to this, the amount of ILP is limited. Combined with the pre-processing loop, a drop in the execution time of this kernel occurs.

### 6.2.3 Exploring the maxfactor option

In Section 5.2, it was mentioned that the maximum possible unrolling factor can be changed. This Section will show the influence of this option of the execution time. Please note that the graphs presented in this Section do not show all kernels, even though they may have a decent speedup. A representative selection of kernels was made. All kernels that are not depicted in the Figures behave similar to kernels that are depicted. The influence of the `maxfactor` option can be seen in Figure 6.2.

![Figure 6.2: Influence of the maxfactor option on the execution time.](image)

For the `fir` and `FracShift` kernels, as well as most other kernels not depicted in Figure 6.2, the higher the unrolling factor, the better the result. However, this does not hold for kernels like the `count_alive` kernel and the `floydwarshall` kernel. Since the goal of these tests was to determine the effect of the maximum unrolling factor, no detailed analysis was done on why some kernels have a low performance. What can be said is that as mentioned previously, these kernels do not perform well due to overhead in the pre-processing that is necessary. However, Figure 6.2 shows that when the maximum
factor is a power of two, the results are improved. This happens because inside the pre-
processing loop, a modulo operation is normally necessary. However, when the unrolling
factor is a power of two, this module is replaced by a logical operator.

But what is the best value for the maxfactor option? To determine this value, a
broader range of values should be tested. Combining this broader range with the knowl-
dge that powers of two give the best results, Figure 6.3 shows the effect of maximum
factors, which are powers of two, on the execution time.

![Figure 6.3: Influence of the maxfactor option as powers of two on the execution time.](image)

The kernels that showed a drop in execution time in Section 6.2.2 show a similar
result here. While the execution time of a single iteration of the pre-processing loop
does not change, more iterations of that loop may be required for higher maximum
factors. Three of these kernels, the bellmanford, floydwarshall and multiply kernels do
show some speedup for low values, but experience a slowdown again for higher values.

The execution time of the FracShift, fir and hamming kernels show a faster result
for a higher maximum unrolling factor, but the speedup reaches a ceiling at a maximum
factor of 32, 16 respectively 16. For the sobel kernel, the difference in speedup between
factors is more extreme, but that kernel also reaches a speedup ceiling.

High values for the maxfactor option cause a more significant slowdown for some
kernels, while for example a maximum factor of 128 does not hold an advantage over a
maximum factor of e.g. 32. Figure 6.4 shows the influence of the maximum factor on
the average speedup over all kernels. This is labelled without-factors.

Even though there were some severe slowdowns, on average the loopunroll engine
does provide some speedup. The best value for the maximum unrolling factor is sixteen,
which corresponds with a previous conclusion that the maximum factor should not be too high, but should neither be too low.

The second plot in Figure 6.4 labelled with-factors, the factors option as mentioned in Section 5.2 is used to limit the possible unrolling factors to powers of two. These results show that the average speedup rises up until the maximum factor of sixteen, just as is the case without using the factors option. However, for maximum unrolling factors higher than sixteen, the average speedup continues to improve slightly, instead of dropping down.

Concluding, the best way for DWARV to determine the unrolling factor is by limiting the unrolling factor to powers of two and, as mentioned in Section 5.2, setting the maximum unrolling factor to a sufficiently high value. That way, the unrolling factor will be determined by the maximum allowed size as long as it is a power of two. Any results mentioned in the rest of this thesis will use these settings.

Using these settings results in an average speedup over all tested kernels of 1.38. In Section 6.2.2 the average speedup of the loopunroll engine with the maxfactor option set to 128 and the factors not used was 1.26. The use of the factors option does indeed provide better results.
6.3 Loop invariant code motion

6.3.1 Introduction

Loop invariant code motion is an optimisation technique that places loop invariant code outside a loop. Loop invariant code is code that is located inside a loop, but does not change between iterations. The implementation details are described in Section 5.3. The improvements made by the loophoist and loopinvariant engines are discussed in Section 6.3.2. Section 6.3.3 will explore different settings for the limit option and Section 6.3.4 will conclude this Section with a comparison between the loophoist engine and the loopinvariant engine.

6.3.2 Evaluation of improvements

In order to evaluate the improvements that are caused by the loop invariant code motion engines, a simple comparison between the baseline design and the design that is augmented with loop invariant code motion is not enough. Different optimisation passes frequently influence each other. A good example of such influence can be seen with the satd kernel. As can be seen in Figure 6.5, applying loop invariant code motion has no influence on its own. However, if loop unrolling is also applied, a small drop in execution time is observed. The speedup is measured against the baseline design, which is DWARV without any of the added optimisations.

![Figure 6.5: Comparison of the loop invariant code motion engines.](image)

When the speedups of all kernels are averaged, loop invariant code motion provides
an average speedup of 1.018 with individual speedups ranging from 0.99 up to 1.28. Combined with loop unrolling, loop invariant code motion provides for an average speedup of 1.40.

While the drop of the satd kernel is insignificant when compared to other improvements, it does serve as a good example of the complexity of interactions between different optimisation passes. What happens is the following: an expression using a two-dimensional array, e.g. \( \text{tmp}[d][2] \) of the data type short, will be translated into pointer arithmetic: \( \text{tmp} + 8d + 4 \). This expression is rewritten by the loopinvariant engine into \( \text{tmp} + 4 + 8d \). These array accesses are used to update a variable that contains some results every iteration of a loop. After the loop is unrolled four times, the unrolled loop will contain four of these statements to update the result-variable. With the rewriting done by the loopinvariant engine, it becomes possible for the exprprop, which expression propagation, to merge these four statements into a single statement. Without the rewriting this is not possible. Finally, the merging done by the exprprop engine causes a larger delay due to scheduling dependencies.

Out of all kernels, only the floydwarshall kernels show a significant speedup when loop invariant code motion is applied on its own. For this kernel, the effect of applying loop-unrolling on top of loop invariant code motion is additive. The floydwarshall kernel experiences a drop after loop unrolling with respect to the baseline. With respect to the speedup of just loop invariant code motion, the same magnitude of dropdown is present. The kernel experiences a speedup of 1.28 when just loop invariant code motion and a speedup of 1.22 when loop invariant code motion is combined with loop unrolling.

The sobel kernel contains loop invariant code, but no speedup occurred when only loop invariant code motion is applied. However, the code motion does expose more parallelism inside the loop, as can be seen from the results where both loop unrolling and loop invariant code motion are applied.

With the multiply kernel, something interesting happens. Both loop-unrolling and loop invariant code motion fail to improve this kernel. Both optimisation methods introduce some overhead. Loop-unrolling requires a pre-processing loop and loop invariant code motion requires a guarding if statement. However, when both techniques are combined, a speedup was measured of 0.87, which is larger than the speedup of applying only loop unrolling of 0.82. The overhead of one pass allows the code to bypass the overhead caused by the other optimisation, causing the kernel to gain a few cycles each time a specific branch operation is executed.

### 6.3.3 Exploring the limit option

In Section 5.3.2.2 the limit option was mentioned. This option determines 'how large' an expression has to be if it is moved outside of the loop. This Section will provide results that show the effect of this option on the speedup.

Figure 6.6 shows that the count_alive and multiply kernels benefit from loop invariant code motion, provided that the limit option is not set higher than three. For the evolve_hw and hw_non_max_supp kernels, the upper limit of three also holds. These kernels do contain loop invariant code, but with the default value of the limit option of four, this code is not expensive enough to justify moving it out of the loop.
6.3. LOOP INVARIANT CODE MOTION

6.3.4 Comparing the loophoist engine against the loopinvariant engine

Up to this section, the combination of the loophoist and loopinvariant engines has been referred to as loop invariant code motion. The difference between the engines is that the loophoist engine performs loop invariant code motion on statements, while the loopinvariant engine performs code motion on expressions. In this Section, the performance of these engines individually will be compared.

Figure 6.7 depicts the results of the comparison. In this Figure, the baseline plot shows the results of DWARV with loop unrolling enabled. From the results, it is clear that both engines cause improvements. The loophoist engine causes an average speedup of 1.004 and the loopinvariant engine causes an average speedup of 1.029. Except for the minor drop for the satd kernel as explained in Section 6.3.2, these engines do not slow
down any kernels. The reason that the *loopinvariant* engine outperforms the *loophoist* engine is due to the fact that in a piece of code, it is more likely that there are loop invariant expressions (small granularity) than statements (larger granularity).

![Figure 6.7: Comparison of the loophoist engine with the loopinvariant engine](image)

### 6.4 Software-sided caching

#### 6.4.1 Introduction

The *cache* and *loopscalar* engines apply software-sided caching, as described in Section 5.4. The number of memory operations are reduced by these engines by replacing read and write operations with accesses to temporary variables. This Section will evaluate the improvements made by these engines.

#### 6.4.2 Evaluation of improvements

To evaluate the improvements due to the *loopscalar* and *cache* engines, both engines are compared in Figure 6.8 with DWARV without any optimisations, the baseline design. The Figure also shows the effect both engines have when combined with the previous optimisations.

The *evolve_hw* and *FracShift* kernels both show a speedup when the *loopscalar* engine is used on its own. On average, the speedup due to the *loopscalar* engine is 1.02. The averaged speedup for just the *evolve_hw* and *FracShift* kernels is 1.18. Interesting to
6.4. SOFTWARE-SIDED CACHING

Figure 6.8: Comparison of the software caching engines.

Note is that the multiply kernel causes a drop in execution time, but this drop does not show up when the loopscalar engine is combined with other optimisations.

The FracShift and floatPoint_IIR7 kernels provide a speedup for the cache engine. With an average speedup of 1.01, the cache engine does not perform as well as the loopscalar engine. However, for none of the tested kernels a drop in execution time was encountered. When these two engines are combined with the previous optimisation engines, an average speedup of 1.45 is achieved. Compared to the speedup of 1.40 for loop unrolling combined with loop invariant code motion, the cache and loopscalar engines work best when combined with those other optimisations.

Except for the multiply and FracShift kernels, nothing interesting happens when the loopscalar and cache engines are combined with the previous optimisations. In the multiply kernel, an access to a local array is cached. On a GPP, a reference to a local array is a memory operation. However, DWARV treats local arrays differently. Instead of storing a local array in memory, the array is stored in a set of registers. Therefore, a reference to a local array is not a time-consuming memory operation and caching a local array does not provide a speedup. The difference between the test where only the loopscalar engine is applied and the test where all optimisations are applied is due to overhead. The caching should only take place if the loop is run. When the loopscalar engine is applied, the introduced overhead causes a drop in execution time. But when loop invariant code motion and loop unrolling are applied, the overhead necessary for the loopscalar engine is already present. That is why there is no further drop in execution time when comparing with the test where the loopscalar engine was not used.
CHAPTER 6. RESULTS

The seemingly non-additive increase of the speedup of the FracShift kernel happens because the improvements caused by the loopscalar and cache engines are multiplied by the loopunroll engine. The improvements are copied when the body of the unrolled body is generated, causing the extra speedup.

6.5 Algebraic simplifications

6.5.1 Introduction

As discussed in Section 5.5, the algebraic engine optimises the IR by rewriting expressions. By reordering subexpressions and by using well known algebraic relations, this engine improves the resulting design. This Section will evaluate and discuss the improvements made by this engine.

6.5.2 Evaluation of improvements

The results of the evaluation of the algebraic engine can be seen in Figure 6.9. In this plot, the baseline results are those obtained using a version of DWARV without the optimisations. On its own, the algebraic engine causes an average speedup of 1.004 compared to the baseline results. However, when combined with the other optimisations, the average speedup between all optimisations and all optimisations but the algebraic simplifications becomes 0.999. All in all, the algebraic engine did not provide the speedup that was expected. The average speedup of all optimisations, including the algebraic engine, is 1.447.

The satd and sobel kernels both experience a drop due to the algebraic engine. Since the algebraic engine balances expressions, dependencies also change because of this. This is what happens with the satd kernel. An expression is rewritten and a variable which contains the same part of that expression in both tests has different dependencies. In the baseline version, the variable is dependant only on other variables. However, in the version where the algebraic engine is applied, the variable becomes dependant on a memory reference. This causes the extra delay for the satd kernel.

The effects of the algebraic engine on the sobel kernel are similar. The engine rewrites an expression similar to \(-1-x\) into \(-(1+x)\). Mathematically, the two expressions are the same. However, for DWARV a single subtraction is changed into an addition, followed by a negation. An extra cycle is introduced because of this. At first glance, a single cycle may not sound that problematic. However, when this cycle is added inside a nested for-loop, as is the case in the sobel kernel, the single cycle becomes more serious. That it is more serious is shown by the dropdown in execution time.

The floydwarshall and fir kernels show no drop in execution time when just the algebraic engine is applied. But when combined with the loopunroll engine, there is a drop, as is seen in Figure 6.9. The algebraic engine rewrites expressions, but on itself no speedup is presented. However when multiple loop bodies are combined, these rewrites change the result of the scheduling. Because of this, the calculation of a memory address takes an additional cycle, causing the longer execution time.
6.6. ENGINE ORDERING

With the evolve_hw kernel, the following happens: when the algebraic engine is applied a number of expressions are being rewritten. When the IR is scheduled, these rewritten expressions can be scheduled more efficiently, causing a speedup. When the other optimisations are applied, the rewritten expressions do not influence any other engine. The result of all optimisations is therefore the effect of the algebraic engine and the effects of the other engines combined, as can be seen in Figure 6.9.

6.6 Engine ordering

6.6.1 Introduction

In Section 2.3 it was mentioned that determining the phase ordering of optimisations is very complex. The ordering of the optimisation engines in this thesis was determined by examining and predicting the effect of engines on each other. This Section will discuss a full space exploration of four optimisation engines in order to validate the ordering designed in this thesis.

Section 6.6.2 will shortly describe the setup that was used to perform the space exploration. Section 6.6.3 will discussed the results averaged over all kernels and Section 6.6.4 will remark on the effect of the engine ordering on individual kernels.
6.6.2 Experimental setup

To determine the effect of enabling or disabling optimisations, four engines where chosen and tested in different orderings. The engines that were chosen are the constant evaluation and propagation engines, the loopscalar engine, the loopinvariant engine and the loopunroll engine. These engines were chosen because of the speedups caused. Each of these engines has managed to speedup at least some kernels. The set of engines was limited to four, because testing all possible combinations of four engines is still manageable.

The four engines were placed together in different orderings, but before the other optimisation engines. Where necessary, mainly after the loopscalar and loopunroll engines, instances of the loopanalysis engine were placed. This was done when an engine that requires the loop information was placed after one of the two engines that can invalidate the information provided by the loopanalysis engine.

6.6.3 Average results

The results of the engine ordering tests are shown in Figure 6.10. In this Figure, the results of the various kernels are averaged. The speedup displayed is with respect to the following ordering: constant evaluation and propagation, the loopscalar engine, the loopinvariant engine and the loopunroll engine. This ordering was chosen because it is the one that is closest to the ordering determined in Chapter 5.

![Figure 6.10: Average speedup of different orderings of four of the optimisation engines. 'c' indicates the constant evaluation and propagation engines, 'ls' indicates the loopscalar engine, 'li' indicates the loopinvariant engine and 'lu' indicates the loopunroll engine.](image)
The impact of the engine ordering is minimal. The slowest combination causes an average speedup of 0.97 while the fastest combination has an average speedup of 1.01. While individual kernels can experience a larger speedup or a larger drop in execution time, as discussed in Section 6.6.3, the focus of this thesis is to provide a sequence of optimisations that perform good for a large variety of kernels.

Out of all the combinations of engines, there is only one combination with a similar performance to the ‘original’ that is the c-ls-li-lu order, ordering of engines. This is the ordering where the loopscalar and loopinvariant engines are switched. Only two combinations outperform the ‘original’ combination: the combinations starting with the loopscalar engine and followed by the loopunroll engine. On first sight, the speedup of these combinations can be due to loopinvariant code that is introduced by the loopunroll engine. And after constant evaluation and propagation, the unrolled loop will contain even more loopinvariant code. But if that is the case, then why does the lu-li-c-ls ordering have a speedup of 0.98? This small example illustrates the difficulty of trying to grasp the effects that different kinds of optimisations have on each other. Especially for a large number of different optimisation techniques, methods like those mentioned in Section 2.3 are required to determine an optimal ordering of different optimisation techniques.

6.6.4 Detailed results

Where Figure 6.10 showed only the average of the results of the ordering tests, Figure 6.11 provides more detailed results of the same tests. Like in Section 6.6.3, all speedups are with respect to the c-ls-li-lu order. Furthermore, kernels where speedup varied less than ten percent are excluded in order to keep the Figure readable.

The conclusion that can be drawn from this Figure is that there is no single ordering that works well for all kernels. The orderings that start with applying loop unrolling work well on the satd kernel, providing a speedup with respect to the ‘original’ ordering, which is the c-ls-li-lu order, of over 1.2. Perhaps because the loops in the satd kernel can be fully unrolled. However, the same combination of optimisations cause the multiply kernel to approach a speedup of 0.5. To complicate matters further, the FracShift kernel experience a speedup for the combinations that start with loop unrolling and where constant propagation takes places before loopinvariant code motion. But for the other combinations starting with loop unrolling, the FracShift kernel has a drop in execution time.

This also shows the importance of using a representable set of kernels to determine the optimal ordering of engines. Suppose that all kernels used to determine the optimal ordering have characteristics very similar to the satd kernel. The results from such a test would result in an incorrect conclusion if the sequence of optimisations is applied to other kernels.
6.7 Comparison with LegUp

6.7.1 Introduction

It is interesting to know how DWARV performs with these new optimisations compared to other tools, like those mentioned in Section 2.2. This Section will compare DWARV with LegUp[7]. The reason LegUp is chosen for this comparison is that it is freely available and very similar to DWARV. LegUp and DWARV have similar restrictions, though DWARV provides support for floating point operations while LegUp does not. Both compilers work on specified function, though LegUp can also compile entire programs. Finally, the two targeted architectures are reported to be very similar in [19]. In Section 6.7.2 the number of cycles of DWARV, DWARV with optimisations and LegUp will be compared with each other. Section 6.7.3 will discuss the influence of optimisation techniques on DWARV and on LegUp.

6.7.2 Comparing DWARV with LegUp

From all the kernels that were tested, the radixsort, multiply and satd kernels either did not compile with LegUp, or the simulation results were incorrect. The results for the remaining kernels are shown in Figure 6.12. This Figure shows the speedup of both DWARV with optimisations and LegUp with respect to DWARV without any optimisa-
6.7. COMPARISON WITH LEGUP

Apart from the fir and fft kernels, which always appear faster when using DWARV, and the viterbi kernel, which appears faster when using DWARV with optimisations, LegUp seems to outperform DWARV. However, it is important to note that what is compared are not actual execution times, but the number of cycles in a design. After examining the Verilog code generated by LegUp for the hamming kernel, it turns out that LegUp performs operations that depend on each other within a single cycle. DWARV uses a different approach and will split these operations over two different cycles. It is likely that a design made by DWARV will be able to run at a higher frequency than a design made by LegUp. In [19], it is reported that there are kernels where LegUp produces a faster design in terms of the number of cycles, but where the actual execution time after synthesis is in favour of DWARV. So unfortunately, without synthesis no conclusions can be drawn on which HDL-compiler produces a faster design.

6.7.3 Impact of optimisations

LegUp makes use of optimisation passes built into the LLVM framework [21]. In the context of this thesis, it is interesting to evaluate the impact of these passes on the speedup and to compare those results with DWARV. Figure 6.13 shows both the impact of the optimisations from this thesis on DWARV with respect to DWARV, without any optimisations, and the impact of the LLVM optimisation passes on LegUp, with respect to LegUp without any optimisations. Overall,
the optimisations in LegUp produce a speedup in every kernel, except for the \texttt{fft} kernel. The difference with DWARV is due to the fact that LegUp simply uses more optimisation techniques than DWARV.

![Figure 6.13: Impact of optimisations for DWARV and LegUp](image)

The Figure also shows that there are promising optimisation techniques not yet integrated with DWARV that can produce good speedups. This can be seen in the \texttt{hamming} kernel, the \texttt{mandelbrot} kernel and the \texttt{viterbi} kernel. DWARV either cannot optimise a kernel, as is the case for the \texttt{hamming} kernel, or it optimises the kernel, but not as much as LegUp can.

Another observation based on these results is the kernels for which the optimisations in DWARV achieve a notable speedup. The impact of the optimisations from this thesis on the \texttt{count_alive_mod}, \texttt{evolve_hw}, \texttt{fir}, \texttt{floydwarshall} and \texttt{sobel} kernels is larger than the impact that the LLVM optimisation passes have.

### 6.8 Conclusion

In this Chapter, the six optimisation engines were tested and evaluated. When combining all the optimisations, an average speedup of 1.45 was achieved. The lowest speedup was 0.85 for the \texttt{count_alive} kernel and the largest speedup was 2.8 for the \texttt{sobel} kernel. Three kernels have a speedup lower than one and four kernels have a speedup of over 2.5. The speedup of all other kernels varies between 1.0 and 1.5.

The \texttt{loopunroll} engine is responsible for most of the speedup. When applying only these optimisations, an average speedup of 1.38 was reached. On the contrary, the \texttt{alge-}
6.8. CONCLUSION

The *braic* engine is the least useful optimisation engine. While it causes a minimal speedup on its own, when it is combined with the other optimisations, this engine causes a drop in the average speedup. The other four engines have positive contributions, leading to the 1.45 speedup.

This Chapter also discussed the performance of optimisations in DWARV and a similar tool called LegUp. The impact of the optimisations in both tools is similar for most kernels. For some kernels, the optimisations in DWARV have a higher impact, while for others there is still room for improvements.
In the introductory chapter, an example was given of the effect of how code is written on the execution time. For the satd kernel, a speedup of 3.3 was achieved by manually modifying the C-code. This observation led to the goal of this thesis: introducing a number of optimisation techniques into DWARV to improve the execution time.

Six optimisation engines were integrated with DWARV. The techniques covered by these engines include loop unrolling, loop invariant code motion, software-sided caching of memory operations and algebraic simplifications.

The average speedup achieved by the six engines is displayed in Figure 7.1. The speedup in this Figure is with respect to DWARV without any added optimisations.

![Speedup Graph](image)

Figure 7.1: Average speedup of the different optimisation engines.

These results clearly show that most of the achieved speedup is due to loop unrolling. The other optimisation engines result in a minimal speedup. When all optimisation techniques are combined together, all engines, except for the algebraic engine positively influence each other. The algebraic engine has a negative effect on the combined average speedup and should be removed from DWARV because of that.

Figure 7.1 shows only the averaged speedup. For individual kernels, the speedup
ranges from 0.85 for the \textit{count\_alive} kernel up to 2.80 for the \textit{sobel} kernel. In total, three kernels were slowed down. For four kernels, a speedup of over 2.5 was achieved. Overall, these optimisation techniques achieve good results. However, a comparison with a similar tool has shown that there is room for more improvement.

The \textit{satd} kernel, mentioned in Chapter 1, has a speedup of 1.4, without using the algebraic simplifications. It is not yet at the 3.3 speedup reported there, but it is a step in the right direction.

7.1 Future work

The results achieved in this thesis can be expanded by further research. Proposed topics are:

- The results in this thesis were obtained at simulation-level. For more detailed results and verification that the designs are usable, the optimisation engines should be tested after synthesis has been done.

- Section 6.6 has explored the phase ordering problem with four engines. However, DWARV contains more engines, including but not limited to optimisation engines. For the best results, the techniques of Section 2.3 should be used to explore this search space and determine the best sequence of engines in DWARV.

- The \textit{loopunroll} engine comes with an option to set the maximum size of the unrolled loop body: the \texttt{maxsize} option (Section 5.2). The effect of this option on the resource usage of an FPGA should be studied, or prediction techniques, such as mentioned in [18], should be used to determine the unrolling factor.

- In Section 6.2, it was mentioned that loops containing conditional statements, such as if-statements, are slowed down by unrolling them. Further research can be done on similar characteristics of the code and their effect on optimisation techniques. These results can be used to steer the optimisation process by enabling or disabling optimisations, based on certain characteristics.

- The CoSy framework contains engines for implementing the software pipelining technique. This technique reduces dependencies between iterations, enabling further optimisations for other engines.

- In accordance with the C-standard, if-statements containing a logical AND or logical OR are evaluated from left to right. However, if none of the boolean statements connected by these AND- and OR statements has side effects, the evaluations can be done in parallel, speeding up the design. The work described in Section 5.2.3.1 can be used as a starting point.

- Section 3.5 introduced many optimisation engines. This thesis has focused on the high priority engines. However, the medium- and low priority engines can also improve the results and should be implemented for the best results.


