Optimising Transformations for Hardware Compilation

A program is a spell cast over a computer, turning input into error messages.

– Anon

June 15, 2005

Author: Ashley Brown, ashley.brown@imperial.ac.uk
Supervisor: Prof. Wayne Luk, wayne.luk@imperial.ac.uk
Second Marker: Dr. Paul Kelly, paul.kelly@imperial.ac.uk

Department of Computing, Imperial College London
Abstract

This report presents a meta-language for the transformation of programs written in Celoxica’s Handel-C language. The design of an accompanying prototype transformation engine is discussed, along with an analysis of the effectiveness of the technique.

A set of example transformations are used to illustrate the type of results possible using the approach presented. When starting with basic code, some designs transformed with the example transformations can operate up to 45% faster without the use of additional logic on the target chip.

The main contributions contained in this report are:

- The first transformation language for parallel hardware descriptions with data integrity conditions (Chapter 3),
- A design and implementation of a parallelising transformation system for that language (Chapter 4),
- Automatic transformations capable of achieving 35-70% reduction in execution time, depending on circumstances (Chapter 5),
- An insight into the interaction of transformations, both with each other and with the platform their output runs on (Chapter 6).
Acknowledgements

This project builds on and uses tools created by a variety of people, with input and assistance from many others. My thanks go to the following people who have been directly involved in, or have given ideas for, this project:

- Prof. Wayne Luk, for suggestions on where to focus and for numerous opportunities throughout the years of my degree.
- Dr. Paul Kelly, for suggestions and his enthusiasm for the project.
- Dr. Oskar Mencer, for sharing his thoughts on the direction the project should take.
- Tim Todman for his assistance with the Cobble framework.
- Oliver Pell for his fast responses and solutions to Pebble 5 problems.
- Rob Dimond for his help whilst collecting test results.

And those who have contributed in other ways:

- Karen Osmond for her mastery of \LaTeX.
- David Brown for help with high-quality colour printing.

Special thanks go to Celoxica for loaning me an RC250 development board for testing.

Finally, this project comes at the end of a four year journey, shared with too many people to mention here individually. My thanks go to my friends and family who have given me support, made me laugh or otherwise kept me sane throughout the degree.

Created with \LaTeX, edited with \texttt{edit}
# Contents

## 1 Introduction

1.1 Motivation .................................................. 2
1.2 Aims .................................................. 3
1.3 Tradeoffs in Digital System Design ......................... 3
1.4 Solution .................................................. 4
1.5 Conclusions on the Approach ................................ 5
1.6 Summary .................................................. 5

## 2 Background

2.1 Software Compilers and Transformation Tools .................. 8
2.2 Hardware Compilation ........................................ 10
  2.2.1 Overview ........................................ 10
  2.2.2 Reconfigurable Computing ................................ 11
  2.2.3 EDA Tools Process Notes .................................. 11
2.3 The Handel-C Language ........................................ 12
2.4 Cobble and Haydn-C .......................................... 13
2.5 Functional Languages ....................................... 13
2.6 Aspect Oriented Programming .................................. 14
2.7 Code Transformation Tool .................................... 14
2.8 Available Optimisations and Transformations ................... 16
  2.8.1 Loops ........................................ 17
  2.8.2 Scheduling and Unscheduling ........................ 18
  2.8.3 External Memory Access ............................. 19
2.9 Handel-C Specific Optimisations ............................... 20
  2.9.1 Converting For Loops to While Loops .................. 20
  2.9.2 Converting Sequential Array Walks to Shifts ............ 21
List of Figures

2.1 Abstract Representation of Compiler Passes .............................................. 8
2.2 Tree Matching for Transformations ......................................................... 9
2.3 Stratix DSP Block Dual Multiplier .......................................................... 11
2.4 Constant Elimination in Haskell .............................................................. 15
2.5 Constant Elimination in ctt ...................................................................... 15
2.6 Constant Elimination in CML ................................................................. 15
2.7 Scalar Replacement ................................................................................. 19
2.8 Scalar Replacement - Registers ............................................................... 19
2.9 Conversion of Sequential Array Walks to Shifts ........................................ 21
3.1 Anatomy of a CML Transformation ......................................................... 27
3.2 CML Transformation Data Integrity Conditions ...................................... 29
4.1 Design Space Exploration — Theoretical Design Flow .............................. 38
4.2 The Local Maxima Problem ..................................................................... 40
4.3 Design Space Exploration — Generating Options ..................................... 40
4.4 Automatic Parallelisation of a While Loop with Composite Transformations 42
4.5 Options for Integration with Cobble ......................................................... 43
4.6 Final Solution Overview — Transformation Engine Operates on AST ........ 45
4.7 Cobble-CML Architecture: Additional Compiler Passes ......................... 45
4.8 The Cobble-CML Toolchain ................................................................... 46
6.1 Matrix Multiply Area/Execution Time Comparison for Stratix .................. 59
6.2 Matrix Multiply Area/Execution Time Comparison for Virtex ................... 60
6.3 Dynamic Power Estimate and Execution Time ......................................... 62
6.4 Matrix Multiply Best Execution Time ....................................................... 64
6.5 Matrix Multiply $f_{\text{max}}$ Comparison ..................................................... 65
List of Tables

3.1 Data Flow Sets — Uses and Defs Sets ........................................... 30
3.2 Data Flow Set Syntax ................................................................. 30
3.3 Successful Assignment to Wildcard Entries .................................. 34
3.4 Unsuccessful Assignment to Wildcard Entries ............................... 35

6.1 Number of Cycles for Matrix Multiply ......................................... 58
6.2 Number of Altera Logic Elements or Xilinx Slices for Matrix Multiply .. 61
6.3 $f_{\text{max}}$ for Matrix Multiply ..................................................... 61
6.4 Number of Cycles After Transformation for Other Programs .............. 67
6.5 Logic Usage After Transformation for Other Programs ................. 67
6.6 Execution Time and Logic Usage Results for the Sobel Supporting Test ... 67
Chapter 1

Introduction

*Therefore, since brevity is the soul of wit, And tediousness the limbs and outward flourishes, I will be brief.*

– William Shakespeare

High-level language to hardware design technology provides a faster development model than hand-crafting digital systems. The ultimate aim is to take software code and produce an efficient digital system design.

Celoxica’s Handel-C [3] language is one of the products with this aim, allowing the specification of hardware designs in a C-like syntax. The language removes some features of ANSI C, such as side-effects, but also adds additional constructs for explicit statement-level parallelism.

Due to these syntax changes the language falls short of the ultimate aim. This chapter summarises the problems produced by the Handel-C approach and the contributions made by this project in the pursuit of solving them.

The major contributions made by this report are:

- The first transformation language for parallel hardware descriptions with data integrity conditions (Chapter [3]),
- A design and implementation of a parallelising transformation system for that language (Chapter [4]),
- Automatic transformations capable of achieving 35-70% reduction in execution time, depending on circumstances (Chapter [5]),
- An insight into the interaction of transformations, both with each other and with the platform their output runs on (Chapter [6]).

The rest of this chapter summarises the project as a whole.

Chapter [2] contains the background material referenced in the remainder of this report.
1.1 Motivation

This project is driven by the desire to solve some key problems with the current approach to hardware compilation.

**Problem:** Code written by software programmers produces poor hardware.

The Handel-C approach to hardware design through a high-level language requires careful thought. An experienced Handel-C designer can produce fast hardware occupying a small area on a chip. A software programmer given little additional guidance and trying to write code in pure C generates slow, bloated hardware. The features of the language which cause this are:

- Strict statement timing (each statement takes one clock cycle),
- Explicit parallelisation (the designer/programmer must state what should run in parallel).

These features make the language very powerful when interfacing with external logic which requires strict timing, however when converting algorithms from C they can hinder performance immensely as the full potential is not fulfilled automatically.

**Problem:** Code written by experienced Handel-C designers becomes difficult to read.

Once an experienced Handel-C designer has improved a software algorithm to suit a hardware design, it becomes exceptionally hard to read. Multiple levels of parallelisation and performance-enhancing trickery yields code with a large amount of syntactic clutter and potentially confusing control flow.

Highly optimised code will contain several different methods to shave one or more cycles off of the algorithm, decrease the critical path to improve the maximum clock rate ($f_{max}$), or alternatively to reduce the chip area used.

---

1 The author speaks from experience on this matter!
1.2 Aims

There are many potential ways of solving the problems mentioned here. In his PhD thesis, Tim Todman proposed a transformation meta-language for his Cobble compiler framework, based on Handel-C \cite{6}. This in turn built on the idea of a transformation meta-language for ANSI C \cite{7}.

The basic concept behind this method is to define a pattern which matches an abstract syntax tree in the language. Should this pattern match part of the abstract syntax tree (AST) for a program, the matched portion is replaced by a tree defined in the meta-language.

The effectiveness of this approach when applied to software is detailed in papers related to that work \cite{7,8}, however the potential for success when applied to a high-level hardware description language was unknown.

The primary aims of this project were:

1. Design a user-programmable transformation system for Handel-C,
2. Produce transformations to achieve the same improvements as an experienced Handel-C designer.

There were also some secondary aims, intended to be fed back into the project to improve its effectiveness later. These aims were to use the programmable transformation system to:

1. Gain an insight into the way transformations interact,
2. Understand how transformation results differ between hardware platforms.

Both of these allow the development of heuristics to enhance the decisions of the transformation engine.

1.3 Tradeoffs in Digital System Design

In the pursuit of the aims of the project, it was important to consider the factors which make a hardware design “good” or “bad”. These are:

- Speed,
- Logic usage (chip area required),
- Power.

Which of these is “good” or “bad” depends entirely on the application. In mobile devices, speed may be sacrificed for the sake of conserving the battery. In high-performance applications where logic usage is not an issue, extracting raw speed at any cost in power or logic may be the goal.
1.4 Solution

The solution presented here uses pattern-matching to perform a pre-compilation and transformation pass on a Handel-C program, generating one or more restructured programs offering differing performance characteristics. In essence, this adds basic behavioural features to an inherently structural language. Generating several options allows them to be passed on for further processing by other tools, to determine which is the best solution for the task.

The prototype transformation engine is built on Tim Todman’s Cobble framework. A transformation meta-language based on Todman’s proposal is defined (Chapter 3), then used to produce a set of transformations falling into three categories:

- **Generic** — match on a variety of generic patterns, intended to improve general constructs found in most programs,
- **Domain Specific** — match constructs typically found only in a particular domain, for instance digital signal processing,
- **Application Specific** — match constructs found in a particular application. This allows an application to run faster without compromising the description of the original algorithm in the code.

Pattern matching alone is not sufficient to apply a transformation correctly. For this reason, the transformation language includes a conditions section where additional constraints can be placed on the match, such as ensuring data dependencies are maintained. This forms one of the most important additions to Todman’s original proposal, which had no data-flow integrity checking at all.

One of the main concerns when applying transformations is which order to apply them, or indeed if they should be applied at all, even when there is a match. The most reliable way to determine the correct combination of transformations is to generate all combinations of applications and test them all. This is a time consuming process but one which can be automated and distributed across machines. The prototype implementation allows the generation of all combinations, or a single solution produced by applying as many optimisations as possible.
1.5 Conclusions on the Approach

A variety of transformations are possible with the transformation language and engine design presented here. However, some additions to the language would have provided even greater scope for transformation definition, in particular with regards to parallelisation.

The transformations that could be defined produced impressive performance improvements with straightforward C definitions of algorithms, providing 40-70% decreases in best execution time. These results are presented in Chapter 6.

The key difficulty with the language as it stands is producing a variety of optimisations in a generic form. The examples presented within this report demonstrate the potential of the programmable transformation engine approach, however some of the most powerful transformations were highly targeted at the specific application. Section 7.4.1 discusses additional features which would aid the transformation of these to a generic form.

The generation and testing of all options — design-space exploration (DSE) — permits the user to define several transformations which match the same pattern, but produce different results on substitution. DSE mode permits all matching transformations to be applied, generating multiple separate options. Comparison of the designs then reveals which one provides the better characteristics (in terms of speed, resources, power) for the given situation. An example of this could be restructuring of a 

for loop, with several possible restructuring options giving varied levels of parallelism.

1.6 Summary

This report presents a transformation language and proof-of-concept transformation engine for the restructuring of digital system designs described in the Handel-C high-level language. The aim of the transformation system was to take standard C-style code and allow automatic transformation to efficient hardware-optimised code, whilst preserving the original source.

Testing revealed that dramatic improvements were possible with standard C constructs, in some cases allowing execution in 30% of the time, with execution in 60% of the time being more common.

The final solution had several limitations, however it is possible to overcome some of these with additions to the transformation systems.
Chapter 2

Background

*There is nothing like looking, if you want to find something. You certainly usually find something, if you look, but it is not always quite the something you were after.*

— Thorin Oakenshield in Tolkein’s *The Hobbit*

The art of software compilation has come a long way in thirty years. Mapping software code to a hardware design is still some way behind.

In the software world, compiler frameworks such as SUIF [9] provide a simple way to create custom compilers with application-specific optimisations. Additionally, a large body of work exists in relation to generic and domain-specific [10, 11, 12] optimisations for software compilation.

During the past fifteen years, work has been undertaken to produce electronic circuit designs from high-level programming languages, such as Occam [13]. This “hardware compilation” process introduces additional constraints, requirements and optimisations into the traditional software compilation flow.

This chapter discusses current work in these fields and its application to this project. The discussion will merge some ideas from the software and hardware compilation worlds, before leading into subsequent chapters for a description of the design decisions made during the project.

Section 2.1 discusses current software compilers and transformation tools.

Sections 2.2, 2.3 and 2.4 cover tools for compilation to hardware designs.

Sections 2.5, 2.6 and 2.7 provide information about possible base systems for a programmable transformation engine.

Finally, Sections 2.8 and 2.9 cover general optimisations and those specific to Handel-C, respectively.
2.1 Software Compilers and Transformation Tools

Conventional compilers follow well established principles detailed in several books, the most notable being “The Dragon Book” [14]. Whilst they can be broken into a number of different phases, we concern ourselves with a more abstract view, as shown in Figure 2.1(a).

A conventional compiler typically uses several passes over an intermediate representation (IR, usually a tree of some form) to perform generic and domain-specific transformations and optimisations. These passes can be considered to be separate processes, hand-coded for the specific compiler or compiler framework and IR being used. Traversing the IR is a common characteristic of these optimisations and is an ideal candidate for factorisation out.

Rule-based program transformation systems have been in existence for many years [15], with the recognition that readable code is generally not efficient and vice versa. Burstall and Darlington tackled this problem in 1977, presenting a solution to turn a “very simple, lucid and hopefully correct program” into a more efficient one [16]. More recently, Pettorossi and Prioetti presented an overview of the program transformation methodology when applied to functional programs [17].

CTT [6] is an extensible transformation engine for ANSI C, based on tree pattern-matching. A transformation is defined with a tranformation meta-language, which describes what tree pattern should be matched, what conditions are placed upon the tree and what pattern to generate afterwards.

Figure 2.1: Abstract Representation of Compiler Passes. (a) A compiler can be broadly split into three sections—reading and understanding the input source, performing transformations on the program and generating the output. (b) Adding the programmable transformation engine into the system. When combined with simple data dependency analysis it has been shown that CTT is capable of parallelizing 85% of loops suitable for modern single-instruction multiple-data (SIMD) instruction sets [7]. Figure 2.1(b) shows where the programmable transformation engine fits into a conventional compiler framework. Figure 2.2 illustrates the application of transformations using tree pattern matching.

ROSE is a programmable source-to-source transformation tool used for the optimisation of C++ object-oriented frameworks. Scientific software typically uses highly abstracted C++ classes with overloaded operators to simplify programming. This structure prevents the compiler from carrying out many optimisations, resulting a drop in runtime performance which ROSE attempts to address. A set of transformations is specified, allowing the abstracted representation to be converted into highly optimised C++ code, which is then compiled with
a standard C++ compiler.

Eelco Visser’s Stratego [18] is another framework for specifying program transformations using rewriting rules. It permits the specification not only of rewriting rules themselves, but also transformation strategies - when to apply certain rules and which order to apply them. This is an important part of finding the optimum transformed solution, but as will be discussed later when transformation strategies are moved into the world of hardware compilation they must take many more factors into account.

**Figure 2.2:** Tree matching for transformations.
(a) The source expression is parsed to an AST.
(b) A set of transformations, including this one for zero-elimination, is loaded. The diamond indicates a “wildcard” which will match any expression.
(c) The transformation maps onto the source AST. The wildcard entry becomes the expression 1 and the substitution is performed. The resulting tree is ready for further transformations, for example constant-folding.
2.2 Hardware Compilation

2.2.1 Overview

Hardware compilation takes a high-level source language such as C and produces a hardware circuit description. This hardware description can be realised in silicon with an application-specific integrated circuit (ASIC), or placed into a reconfigurable device such as a field-programmable gate array (FPGA).

There are two distinct approaches used in hardware compilation systems: behavioural and structural.

With the behavioural approach:

- The program is simply a high-level description of the algorithm.
- The compiler uses a set of constraints to implement the algorithm as it sees fit.
- There are no guarantees on the timing of each statement.
- The final hardware can vary greatly depending on the tool and input constraints.
- The compiler has great flexibility in the transformations it can perform.

On the other hand, with a structural specification:

- The timing of the program is well-defined.
- The compiler has little freedom in implementation—the final hardware design must remain faithful to the original description.
- The timing and resource usage of the final hardware design is predictable from the original source code.

An optimal, finished, structural design generally bears little resemblance to the original algorithm being implemented. Straightforward operations such as multiplication and exponentiation can take on dramatically different structures in the search for scalability and performance.

---

1 Behavioural compilation schemes adjust resources and timings based on other factors, such as branch probabilities. [19]
2.2.2 Reconfigurable Computing

Placing the design of a digital system onto silicon is expensive. While an ASIC becomes cheap in volume, as with most things the cost of a one-off build precludes testing by building repeated prototypes. Consequently rigorous verification and simulation takes place to ensure correct operation of a design beforehand.

An FPGA is a user-programmable integrated circuit, capable of implementing a large array of digital system designs. Modern FPGAs typically contain a set of configurable lookup tables, routing logic and specialised components such as multipliers and RAM blocks. For example, embedded digital signal processing (DSP) blocks are found on the Altera Stratix, Stratix GX and Stratix II and Xilinx Virtex II and Virtex 4 devices.

Altera’s Stratix DSP blocks can operate at 420MHz and perform multiply-accumulate and multiply-add operations. Consequently, certain AST patterns can be mapped directly onto a single embedded hardware block in an FPGA. For example, the expression \((a \times c) \pm (b \times d)\) can be mapped into part of one DSP block, as shown in Figure 2.3.

![Partial DSP Block](image)

**Figure 2.3:** Implementation of \((a \times c) \pm (b \times d)\) in the Stratix DSP block dual-multiplier mode.

Using reconfigurable parts such as FPGAs provide several benefits, particularly during prototyping:

- A new design can be loaded in seconds.
- SRAM-based FPGAs may be reconfigured an unlimited number of times.
- In the field, hardware can be “patched” in a similar way to software.
- An FPGA may be reconfigured with a different design to suit the task at hand.

High-level language to hardware compilation technology is often used to target FPGAs, providing fast generation of a hardware design followed by a rapid prototype implementation to hardware.

2.2.3 EDA Tools Process Notes

The design decisions made in this project were driven chiefly by the needs of the current users of similar electronic design automation tools. Users of these tools require a good solution as...
quickly as possible, but with the option of spending more time later in the design process to produce the best solution possible. Hand tuning is undesirable if a tool could do the job instead.

Verification of designs is also an important issue in the field — once designs progress to ASICs rather than FPGAs, the costs of an error can be substantial.

To take these factors into account, any tool for transforming designs requires:

- A fast transformation stage.
- A slower transformation stage to find the best options.
- A consideration of verification techniques for transformations applied to designs.

### 2.3 The Handel-C Language

Celoxica’s Handel-C is a commercial C-based programming language intended for compilation to a hardware description. Cobble introduces some minor changes to the syntax however the languages can be considered the same for the purposes of this report.

Amongst others, Handel-C introduces the following changes to ANSI C:

- No side-effects in expressions,
- Strictly timed statements,
- No recursion in function calls.

Each Handel-C program uses a central clock for statement evaluation. The timing of Handel-C statements is strict.

- Normal statements take one cycle to execute,
- Function calls and macro procedures take a variable number of cycles,
- All other elements (if, while, do...while) take no cycles to execute.

Strict timing makes Handel-C a structural design tool, where the algorithm must be explicitly written for the specific performance/resources/power target. In contrast, a behavioural tool allows the designer to specify the algorithm and the compiler will generate a design with timings to meet other constraints.

Strict timing prevents the Handel-C compiler from parallelising statements, even when it is possible to do so. Fine-tuning Handel-C by hand to achieve more desirable characteristics can destroy the readability of the code.

---

3 A Handel-C design may have more than one clock, however each clock domain is defined by a separate “main” function.
2.4 Cobble and Haydn-C

The Cobble compiler framework [5, 21] provides a user-customisable system for compilation of C-based source code to hardware, using a base language modelled on Handel-C. Cobble permits the following customisations:

1. the input language can be extended,
2. custom transformation steps can be added,
3. the hardware design generation stage can be extended or replaced.

Todman has also proposed a meta-language for Cobble called CML [5, 21]. This was envisaged as similar to the Boekhold’s transformation engine [6], but with the language translated to C++ and then compiled. The meta-language was intended to have greater scope than simply transformations (item 2 above), but to permit all the extensions of the framework to be described. Section 2.7 covers CTT/CML in more detail.

Haydn-C [22] is a component-based language with C-like syntax that seeks to combine behavioural and structural synthesis. Haydn-C source can be transformed from a behavioural description to a variety of structural descriptions (e.g. pipelined or sequential versions) using a flexible timing model. During hardware synthesis and simulation, a strict cycle-accurate timing model similar to the one in Handel-C is used.

2.5 Functional Languages

Many functional languages [23], such as Haskell [24] and the ML [25] family, include pattern matching as a core part of the language. Because of this, navigating and restructuring trees is a relatively simple task. For the rest of this report, we consider “functional programming languages” to only include the subset with pattern-matching features.

A functional-language based meta-language could implement tree pattern matching in an elegant manner, however it suffers two distinct disadvantages:

- The tree pattern to match must be described explicitly (see Figure 2.4).
- Any but the most basic of transformations lead to complicated code in a style unfamiliar to programmers used to imperative languages.

A meta-language for defining transformations should be simpler than the corresponding C++ implementation, otherwise nothing is gained from using it. A functional language description of a complex transformation can become difficult to understand, so does not fulfil this requirement.

Figure 2.4 shows how the elimination of the constant ’0’ would appear in Haskell.
2.6 Aspect Oriented Programming

Aspect Oriented Programming [26] is an attempt to extend separation of concerns in software development to aspects which permeate throughout typical code, such as logging.

Rather than, for example, adding logging code to every object in an object-oriented system, the code is written without it. A set of aspects define where to insert logging code, for example whenever a file is opened or closed. The standard code is combined with the aspects by a code weaver (an aspect-oriented compiler [26]), to create the final code to be compiled.

AspectJ [27] is Xerox’s implementation of aspect oriented programming for Java. It allows three types of code insertion:

- Before — insert aspect code before matched code.
- After — insert aspect code after matched code.
- Around — replace the matched code with the aspect code (to retain the matched code, it must form part of the aspect code).

“Around” provides an AST replacement mechanism. When describing optimisations it is typical to replace an entire code block, so “Before” and “After” are of little use here.

The code matching and replacement features of Aspect Oriented Programming offer a powerful solution to the problem of separation of concerns with “tangled” code. The same principles could be applied for optimisations rather than code separation, by matching an unoptimised section of code and replacing it with an optimised one. This is the basis of all the options presented here, however the AspectJ syntax provides a more powerful mechanism for matching code fragments, defining many constraints on how to match a piece of code, taking into account control flow.

One major drawback of a solution based on the AspectJ syntax is its complexity. While it provides many useful features and a powerful matching engine, some of these are superfluous and complicate the syntax more than is necessary for optimisations.

2.7 Code Transformation Tool

Code Transformation Tool [6] was a solution developed many years ago and suggested as a basis for the transformation part of the Cobble Meta-language (CML) put forward by Todman [5], but never implemented.

The major advantage of the CTT/CML approach is the simplicity of the syntax. A section of code to match is defined by entering the C code directly, with additional generic elements to match, for example, “any expression”. Furthermore, the CTT syntax has a conditions section to place further constraints on the match such as data-dependency checking, although this is not present in the original CML definition.

Figures 2.5 and 2.6 show how the elimination of the constant ‘0’ would appear in the CTT transformation language and CML.
removeZero :: Expr -> Expr
removeZero (Expr (Binop Plus (Const 0) x)) = x

**Figure 2.4:** Constant Elimination in Haskell. The representation in the functional language Haskell is much more concise, although this is a very simple example. The problem with using a functional language becomes clear—to someone unfamiliar with functional programming the style is unusual. Constructing trees for complex transformations in this way also leads to code which is difficult to read.

```
PATTERN { 0 + EXPR(1) }
CONDITIONS {}
RESULT { EXPR(1) }
```

**Figure 2.5:** Constant Elimination in ctt. Although a simple example, it demonstrates the use of C-based syntax, which is converted to a tree for matching using a parser derived from the standard C. EXPR(1) illustrates the use of *meta-elements* to specify generic patterns.

```
custom_transform {
  pattern {
    0 + expr(1)
  }
  generate {
    expr(1)
  }
}
```

**Figure 2.6:** Constant Elimination in CML. The relation to ctt can clearly be seen, although the CONDITIONS section is absent. Here, expr(1) is a meta-element. The structure is slightly different, as CML is envisaged to support a number of different compiler enhancements in addition to transformations.
2.8 Available Optimisations and Transformations

Conventional software compilers can typically optimise for speed or code size—increasing speed traditionally led to increased size (e.g. loop unrolling and function inlining).

This size/performance tradeoff is also seen in hardware compilation, along with another factor: power consumption. Therefore, the following factors must be taken into account when designing any optimisations:

- The amount of resources/area used on the chip,
- The maximum clock-rate, $f_{max}$ of the design,
- The latency of the design,$^4$
- The power consumption and heat dissipation of the design.

As an example, pipelining can increase $f_{max}$, whilst decreasing power dissipation $^{[28]}$. However, it also increases resource usage and latency through the circuit. Increased parallelism increases resource usage, increases power consumption and may decrease $f_{max}$. The advantage however is reduced latency which can outweigh the $f_{max}$ reduction.

The increasing use of generic microprocessors for multimedia applications triggered the introduction of specialised instructions such as Intel’s MMX instruction set. Single-instruction multiple-data (SIMD) instructions allow one instruction to operate on a vector of data, providing a form of parallelism. CTT has been shown to work effectively for automatic vectorisation $^{[3]}$. A similar optimisation in hardware might, for example, introduce a parallel block with hardware duplication.

Bacon et al. $^{[29]}$ give a comprehensive overview of compiler optimisations. A selection is presented here, however the reader is advised to refer to that paper for further information.

---

$^4$Latency and $f_{max}$ together determine the overall “speed” factor from software compilation.
2.8.1 Loops

There are a variety of loop-related optimisations which could be suitable candidates for pattern-matched optimisations [30, 31, 29]; indeed, the SIMD optimisations are loop-related. The summary below describes a selection of the various loop optimisations available.

- **Loop Unrolling** - The body of a loop is replicated to reduce loop overhead in terms of time and resources, whilst potentially exposing further opportunities for optimisation. Determining when it is correct to unroll a loop and how much to unroll it requires careful modelling of results, with a large number of open issues [32].

- **Loop Interchange** - the order of loops is changed to better match system characteristics, for example cache behaviour. It still has relevance in hardware compilation, as it can reveal possibilities for vectorisation.

- **Loop Fusion** - independent loops with the same bounds are merged into a single loop, allowing them to proceed in parallel.

- **Loop Fission** - a single loop with multiple computations is split into independent loops. This can expose possibilities for vectorisation.

- **Strength Reduction and Induction Variable Elimination** - An operation within a loop is replaced by an equivalent but less expensive operation. For example, given a loop variable $i$ and constant $c$, the expression $c \times i$ within a straightforward 0..n loop may be replaced with a new temporary variable $T$, which is incremented by $c$ on each iteration. If this is the only use of $i$ within the loop, $i$ can be removed entirely and the loop bounds modified to use $T$ instead.

The loop optimisations outlined above require additional information beyond the pattern of the AST, such as data-dependencies and the presence of any statements which disrupt the loop, such as `continue` or `break` in C. Using dataflow analysis to build a dataflow graph allows potential loop optimisations to be identified more easily than with a straight AST [33].

The Polytope Model

The *polytope* model provides a path for the parallelisation of a strict subset of nested for-loops [34, 35]. Loops represented in the polytope model can be converted to a systolic array and exploit parallelism, providing they meet some strict criteria, paraphrased from Griebel:

- The loops must form a perfect loop nest, with assignments only in the body of the inner-most loop.

- Loop bounds must be affine expressions using only indices of enclosing loops and constant parameters such as the problem size.

- The only supported data-structure is the array; array indices have the same conditions as loop bounds. Scalars are treated as 0-dimensional arrays.

- Dependencies must be identical in all iterations.
A program (or program loops) meeting these requirements can be converted to a model-based representation, parallelism extracted and a target program produced from the model. The University of Passau has a loop paralleliser called LooPo to do source-to-source transformations in this way.

2.8.2 Scheduling and Unscheduling

The “scheduling” and “unscheduling” techniques used in Haydn-C [22] and described as an addition to Cobble [21] rely on the conversion of a data flow graph (DFG) to an AST and vice versa. Implementing these transformations from a meta-language would require the analysis or pattern matching of the DFG and a corresponding transformation to an AST. One DFG may produce several ASTs, each with different size, speed and latency characteristics. Consequently, Haydn-C style transformations must be directed to the most desired solution. Any pattern-matching solution would require several output ASTs based on a single DFG, with a mechanism for choosing which one to apply.
2.8.3 External Memory Access

As with microprocessors, external memory accesses from FPGAs are expensive. However, unlike microprocessors FPGAs typically have no dedicated cache in their memory hierarchy. Instead, blocks of fast RAM are included within the FPGA fabric for localised storage.

Scalar replacement modifies loops to place commonly-used variables or array accesses into registers close to the processor, rather than fetching them from memory on each iteration. A similar approach can be used in hardware compilation. The embedded RAM blocks are an alternative to using registers in FPGAs [36], particularly when a large amount of registers would be required.

For example, consider Figure 2.7. \( c[k] \) is used on every iteration of the innermost loop. The entire \( c \) array is traversed on every iteration of the \( j \) loop, \( i \times j \) times. A cache on a microprocessor could avoid reloading \( c \) from memory every time and instead keep it locally. In an FPGA, we would create a local register array of size \( MAXK \). Figure 2.8 illustrates the replacement process.

```c
for (int i = 0; i < MAXI; i++) {
    for (int j = 0; j < MAXI; j++) {
        for (int k = 0; k < MAXK; k++)
            a[i][j] += c[k];
    }
}
```

Figure 2.7: Scalar Replacement

```c
register int r_c[MAXK];
memcpy(r_c, c, MAXK); // copy results
for (int i = 0; i < MAXI; i++) {
    for (int j = 0; j < MAXI; j++) {
        for (int k = 0; k < MAXK; k++)
            a[i][j] += r_c[k]; // local register version
    }
}
```

Figure 2.8: Scalar Replacement

If integers are considered to be 32-bit, we only need \( MAXK \) to be 16 to fill an M512 block [37] on a Stratix device. Using a memory block for this purpose uses less area for a comparable performance than creating 16 32-bit registers.

---

5The M512 is a 512-bit embedded memory block. A Stratix FPGA also has M4K (4096-bit) and M-RAM (512Kbits) blocks.
2.9 Handel-C Specific Optimisations

The compilation scheme used by Handel-C and Cobble lends itself to certain optimisations which can provide excellent performance improvements.

2.9.1 Converting For Loops to While Loops

A minor optimisation which can have a dramatic effect involves rewriting for loops to while loops, then parallelising. Consider the following toy example:

```handel-c
for (i = 0; i < n; i++)
    a[i] = i;
```

This may be re-written as:

```handel-c
i = 0;
while (i < n) {
    a[i] = i;
    i++;
}
```

The timing semantics of Handel-C dictate that reading a variable is performed at the start of a clock cycle and the write takes place at the end. Hence it is possible to read and write a variable in the same cycle and read the old value. So the while loop may be rewritten as:

```handel-c
i = 0;
while (i < n) {
    par {
        a[i] = i;
        i++;
    }
}
```

The optimisation in this case would save $n$ cycles – one cycle per iteration is removed.
2.9.2 Converting Sequential Array Walks to Shifts

When targeting hardware—FPGAs in particular—congestion around a register has a negative impact on resource usage and $f_{max}$. This can occur particularly in loops which walk over arrays, where values from every item in the array are generally read into a single register for further processing during the iteration.

Reading values into one register from multiple places causes congestion, which in turn makes signal routing difficult. Where only sequential array walks are required a more efficient solution is available. The array can be treated as a large rotating shift-register, shifting out values sequentially. This only requires signals to be routed between adjacent registers in the array, between the first and last item and the first item and input/output register.

Figure 2.9 illustrates the problem and solution.

![Diagram](image)

**Figure 2.9:** Conversion of Sequential Array Walks to Shifts - (a) An indexed array causes all data to flow through a single congestion point into a register or other hardware block within the loop. (b) If only sequential access is required, the array may be shift-rotated to provide a single exit point with no congestion. Rotating preserves the order of the array at the end of the operation if the original loop has $n$ iterations, where $n$ is the size of the array.
2.10 Summary

This chapter has presented the background information required to understand the remainder of this report. The basics of compiler optimisations and hardware compilation tools provide an insight into what transformations could be achieved and how the transformation system presented here fits in.

The Cobble Meta-language, Functional Languages and Aspect Oriented Programming all provide a suitable basis for a transformation language, with the reasoning behind the final decision being shown in Chapter 3.

A wide variety of powerful optimisations are possible in software, with many translating well into hardware. In particular, transformations for the parallelisation of loops for SIMD processors can be used. The nature of the base language for this project, Handel-C, provides a number of additional optimisations not seen in software compilers.
Chapter 3

Transformation Language

C: You shoot yourself in the foot.

C++: You accidently create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical care is impossible since you can’t tell which are bitwise copies and which are just pointing at others and saying, “that’s me, over there.”

– Unknown

The transformation language lies at the core of this project. It was important to strike the right balance between power and simplicity, whilst still bearing in mind the time available.

This chapter covers the requirements and design of the transformation definition language for the transformation system presented here.

Section 3.1 covers the requirements of the language.

Section 3.2 covers the foundations of the language and modifications to those foundations.

To illustrate how the language works, Section 3.3 covers an example.
3.1 Requirements

The transformation language must:

1. Provide a syntax accessible to Handel-C users (Section 3.1.1),
2. Support generic, domain-specific and application-specific transformations (Section 3.1.2),
3. Allow the specification of data-integrity conditions (Section 3.1.3).

Section 3.2 provides a corresponding list of how these numbered requirements are met.

3.1.1 Ease of Use

The conventional method of adding an optimisation pass to a compiler is to write it directly, often in C++. For generic transformations this is perfectly acceptable, as the native code can be fine-tuned for additional performance. However, for a user wishing to transform a specific part of their code to another format this is not a practical approach. They may not even know the language of the compiler, let alone be capable of writing a custom transformation in it.

The transformation language must provide a syntax accessible to the user of the language it transforms, Handel-C. This means a familiar syntax with a simple method of defining a transformation.

The core to meeting this requirement is the definition of the base language (Section 3.1.4).

3.1.2 Types of Transformations

There are three types of transformation to consider:

- **Generic** — match on a variety of programs, intended to improve general constructs. Examples include:
  - Automatic parallelisation,
  - Converting counting for loops from < to ==.

- **Domain Specific** — match constructs typically found only in a particular domain, for instance digital signal processing. An example is:
  - Parallelisation or pipelining of multiply-accumulate blocks in digital signal processing designs.

- **Application Specific** — match constructs found in a particular application. This allows an application to run faster without compromising the description of the original algorithm in the code.

The language must permit the specification of all three types of transformation.
3.1.3 Maintaining Data Integrity

The designer expects that the final circuit produced by a compiler will be a faithful representation of the high-level description they provided. Any optimisations must preserve the meaning of the original program, or change it only in restricted ways with which the user agrees.

Performing transformations on the AST essentially modifies the input source, so in a fully automatic transformation system it is required that the algorithmic behaviour after transformation is the same as that before.

Restructuring the AST of a program allows the original meaning to be lost, if care is not taken to ensure the control and data flow is preserved. A vital component of a transformation meta-language is a way to define constraints which ensure the integrity of both control-flow and data-flow. Important examples of how lack of integrity can cause problems are:

- Parallelising two statements where dependencies exist between them causes incorrect results,
- Restructuring a loop with unsafe control flow, such as `break` or `continue` statements, may alter the control flow in subtle ways and cause undesirable behaviour.

3.1.4 Foundations

Chapter 2 discussed previous work suitable for user-specified, automated transformations of source code. The key ones of interest were:

- Functional Programming Languages (Section 2.5),
- Aspect Oriented Programming (Section 2.6),
- Code Transformation Tool/Cobble Meta-language (Section 2.7).

The functional approach provides easy pattern matching, but with a complicated syntax. Achieving an overview of data integrity conditions is also difficult. Many Handel-C users would be unlikely to successfully write a complicated transformation in a functional language.

An aspect-oriented approach with an AspectJ-style syntax could provide a powerful mechanism for defining which parts of the code to replace. The syntax is familiar enough to be accessible to Handel-C users, however it is more complicated than necessary for the purposes of this transformation system.

CTT/CML has a major advantage in the simplicity of its syntax. By defining the pattern to match by writing code in Handel-C directly, the learning curve is far more desirable than the other options.

Todman’s original proposal for CML built on the CTT approach, however removed one of the most important sections — the conditions section. Under CTT, this allowed a variety of data integrity checks to take place. The CML approach would not allow this, leaving many useful transformations without valid descriptions. The language design given here re-instates the conditions section with a new syntax.
3.2 Language Design

The transformation language defined here has:

1. A syntax similar to Handel-C, making it accessible to Handel-C users (Section 3.2.1),
2. Wildcard expressions and statements to support generic transformations, with precise matching available for domain-specific and application-specific transformations (Section 3.2.2),
3. A variety of data integrity checks (Section 3.2.3).

This section covers the final design of the transformation subset of CML (T-CML). This is a variation of CTT and expansion of CML, which was only partially defined by Todman and never implemented.

Complete examples of the language may be found in Figure 3.2, Chapter 5 and Appendix A. Section 3.3 works through a parallelisation example.

A partial EBNF description of the language syntax can be found in Appendix B.

3.2.1 Language Components

Building on the CTT/CML language gives a transformation definition with the format shown in Figures 3.1 and 3.2. Three distinct sections are present:

1. pattern — the code fragment to match,
2. generate — in the event of a match, the code to substitute,
3. conditions — data integrity conditions to check before applying the transformation.

The following changes have been made to the basics of the CML approach:

1. Reinstatement of a revised conditions section,
2. Addition of an always keyword to instruct the transformation system to always apply a transformation when in design-space exploration mode (See Section 4.1.1),
3. Specification of wildcard matches with a name (cmlexpr(init)) rather than a number (cmlexpr(0)) for increased readability.
4. Introduction of a name for transformations, so they may be easily identified for reporting to the user.
Wildcards, such as cmlexpr, allow a pattern to be matched and substituted into the new tree. The pattern section describes the format of the code to match for this transformation. The generate section describes the code which should replace the pattern. Each transformation can have a name to identify it for reporting. CML transformations are defined within transform blocks. The optional always keyword indicates that this transformation should always be applied where it can.

```
// 1 * x = x
always transform std_times1_elim {
  pattern {
    1 * cmlexpr(operand)
  }
  generate {
    cmlexpr(operand)
  }
}
```

Figure 3.1: Anatomy of a CML Transformation
3.2.2 Supporting Generic Transformations

The simple example in Figure 3.1 represents a generic transformation. The language supports generic matches through the use of the following wildcard statements and expressions:

- cmlexpr — matches any Handel-C expression,
- cmlstmt — matches any single Handel-C statement,
- cmlstmtlist — matches zero or more statements.

Handel-C supports parallel and sequential replicators, which are similar to for loops but are executed at compile time. A sequential replicator is essentially a for loop which is always unwound by the compiler, while the parallel replicator executes all of the iterations of the unwound loop in parallel. The use of these within patterns provides additional support for generic matches, as variable length code can be generated based on the pattern match.

Without wildcards the transformation can match code fragments exactly, providing support for domain-specific and application specific transformations.

The example in Section 3.3 demonstrates the use of generic matches with statements.
3.2.3 Data Integrity

The optional conditions block is a new addition to CML and allows enforcement of data integrity constraints:

```
transform auto_par {
  pattern {
    cmlstmtlist(preamble);
    cmlstmt(par1);
    cmlstmt(par2);
    cmlstmtlist(postamble);
  }
  generate {
    cmlstmtlist(preamble);
    par {
      cmlstmt(par1);
      cmlstmt(par2);
    }
    cmlstmtlist(postamble);
  }
  conditions {
    // don't assign to the same place
    defs(cmlstmt(par1)); & defs(cmlstmt(par2)); == {};
    // second statement not waiting on first
    defs(cmlstmt(par1)); & uses(cmlstmt(par2)); == {};
  }
}
```

Figure 3.2: CML Transformation Data Integrity Conditions

The following conditions are supported:\footnote{There is also a “debug” condition which always returns true, but prints the tree passed to it.}

- Data-flow Sets
- Expression Evaluation
- Constant Checking
Data Flow Sets

The most important type of condition for enforcing data integrity is the data-flow set condition, as illustrated in Figure 3.2. The CML approach did not have a conditions section at all, however it is vitally important to ensure the validity of transformations.

The two most important components are the “defs” and “uses” sets for each type of statement. The “defs” set contains variables that are assigned to during a statement, while “uses” contains those a statement uses during execution. A statement in this case may also be a block of statements, in which case the union of the sets for each statement is used.

The example in Section 3.3 illustrates how these can be used to ensure data integrity during transformations. Table 3.1 shows “defs” and “uses” sets for a selection of statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>defs</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1 )</td>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>( x = a + b )</td>
<td>( x )</td>
<td>( a, b )</td>
</tr>
<tr>
<td>( \text{par} { x = a + b; y = c + d; } )</td>
<td>( x, y )</td>
<td>( a, b, c, d )</td>
</tr>
</tbody>
</table>

Table 3.1: Data Flow Sets — Uses andDefs Sets

A small set-manipulation language is defined, along with some standard sets for data-flow. Table 3.2 describes the components of this set language.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Set equality comparison.</td>
</tr>
<tr>
<td>!=</td>
<td>Set inequality comparison.</td>
</tr>
<tr>
<td>{a,b}</td>
<td>A set containing variables a and b.</td>
</tr>
<tr>
<td>{}</td>
<td>The empty set.</td>
</tr>
<tr>
<td>( \text{defs(statement)} )</td>
<td>The set of variables statement assigns to.</td>
</tr>
<tr>
<td>( \text{uses(statement)} )</td>
<td>The set of variables statement uses.</td>
</tr>
<tr>
<td>&amp;</td>
<td>Set intersection, ( \cap ).</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Data Flow Set Syntax

Expression Evaluation

The eval condition evaluates a constant expression passed into it, returning false if the expression is zero or true if not. Wildcards may be passed into the condition, so \( \text{eval(cmlexpr(init) >= 0)} \) would return true if the expression which matched to \( \text{cmlexpr(init)} \) was greater than or equal to zero.

This is an important condition when restructuring loops, to ensure that the loop iteration method is as expected (e.g. the loop starts from zero and counts upwards).
The example transformation given below matches any multiplication expression. The `eval` condition only permits a replacement to take place if one side of the multiplication is a zero, in which case the entire multiplication is replaced by zero.

```
transform
{
    pattern
    {
        cmlexpr(lhs) * cmlexpr(rhs)
    }
    generate
    {
        0
    }
    conditions
    {
        // if either side is 0, generate 0
        eval(cmlexpr(rhs) == 0 || cmlexpr(lhs) == 0);
    }
}
```

**Constant Checking**

Some optimisations are only possible when constants are present in the match. The wildcard expression `cmlexpr` will match any expression, whether it is constant or not. These matches can be passed into the `constant` condition where necessary.

The following very simple example illustrates this:

```
transform
{
    pattern
    {
        0 * cmlexpr(rhs)
    }
    generate
    {
        0
    }
    conditions
    {
        constant(cmlexpr(rhs));
    }
}
```

In this case, anything multiplied by 0 will be replaced by 0, but only if it is a constant. The main use is of course with more complicated examples, such as ensuring that loop bounds are constant in order to automatically parallelise an entire loop. See Section [A.2.1](#) for an example.
3.2.4 Definition of Variables

Transformations need to both match variables and create new ones. The `cmlvar` keyword is used at the top of a transformation definition to define the list of variables to be matched by name. Any variables not matched, but which appear in the `generate` section, are allocated when the transformation is applied.

3.2.5 Future Extensions

The language design presented here reflects those parts which were successfully implemented in the prototype. Experience with the finished transformation engine revealed some important additions which would be required to make it more effective at automatic parallelisation and pipelining transformations. These additions are discussed in Section 7.4.
### 3.3 Example

This section provides an example of the transformation process when used for automatic parallelisation. It demonstrates wildcard matches and data-flow conditions.

The following is an automatic parallelisation transformation, as found in Figure 3.2 and Chapter 5:

```plaintext
transform auto_par {
  pattern {
    cmlstmtlist(preamble); // statements before
    cmlstmt(par1); // first statement
    cmlstmt(par2); // second statement
    cmlstmtlist(postamble); // statements after
  }
  generate {
    cmlstmtlist(preamble);
    par {
      cmlstmt(par1);
      cmlstmt(par2);
    }
    cmlstmtlist(postamble);
  }
  conditions {
    // don’t assign to the same place
    defs (cmlstmt(par1)) & defs (cmlstmt(par2)) == {};
    // second statement not waiting on first
    defs (cmlstmt(par1)) & uses (cmlstmt(par2)) == {};
  }
}
```

Consider also the following code fragment, which the transformation will successfully match:

```plaintext
q = a << 1;
qup = q + 1;
qmn = q - 1;
```
3.3.1 Successful Matching

Table 3.3 lists a possible set of assignments made to the wildcard matches.

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>preamble</td>
<td>( q = a \ll 1; )</td>
</tr>
<tr>
<td>par1</td>
<td>( qp = q + 1; )</td>
</tr>
<tr>
<td>par2</td>
<td>( qm = q - 1; )</td>
</tr>
<tr>
<td>postamble</td>
<td>empty</td>
</tr>
</tbody>
</table>

Table 3.3: Successful Assignment to Wildcard Entries

Following a successful match, the conditions must be verified, by substituting real statements for the wildcards and generating the data-flow sets:

```c
// initial conditions
defs(cmlstmt(par1)); &defs(cmlstmt(par2)); == {};
defs(cmlstmt(par1)); &uses(cmlstmt(par2)); == {};

// substituting the matches
defs(qp = q + 1;) &defs(qm = q - 1;) == {};
defs(qp = q + 1;) &uses(qm = q - 1;) == {};

// applying the sets
{qp} & {qm} == {};
{qp} & {q} == {};

// performing set intersection
{} == {}; // true
{} == {}; // true
```

In this case, the conditions hold so the match is successful. The following code is generated:

```c
q = a \ll 1;
par
{ q = q + 1;
  q = q - 1;
}
```
3.3.2 Unsuccessful Matching

There is a second possible assignment to the wildcard matches given the fragment of code supplied. Table 3.4 illustrates this assignment.

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>preamble</td>
<td>empty</td>
</tr>
<tr>
<td>par1</td>
<td>q = a &lt;&lt; 1;</td>
</tr>
<tr>
<td>par2</td>
<td>qp = q + 1;</td>
</tr>
<tr>
<td>postamble</td>
<td>qm = q - 1;</td>
</tr>
</tbody>
</table>

Table 3.4: Unsuccessful Assignment to Wildcard Entries

Checking the conditions with this assignments leads to a different result:

```c
// initial conditions
defs(cmlstmt(par1)); & defs(cmlstmt(par2)); == {}; 
defs(cmlstmt(par1)); & uses(cmlstmt(par2)); == {};
```

```c
// substituting the matches
defs(q = a << 1;) & defs(qp = q + 1;) == {}; 
defs(q = a << 1;) & uses(qp = q + 1;) == {};
```

```c
// applying the sets
{q} & {qp} == {}; 
{q} & {q} == {};
```

```c
// performing set intersection
{} == {}; // true
{q} == {}; // false!
```

The conditions fail, so no new code is generated. A successful match would have produced the following:

```c
par
{
    q = a << 1;
    qp = q + 1;
}
qm = q - 1;
```

As the second statement is dependent on the first, `qp` would receive the *old* value of `q + 1`, rather than the new one defined in the first statement.

For further information, see Figure 2.2 which shows the matching and replacement process in a graphical form.
3.4 Summary

This section described the requirements for the transformation language, the options available
as a basis for the language and the final design decisions. The key syntactic elements and
data integrity conditions were covered with annotated diagrams.

A worked example illustrated the importance of checking data-integrity conditions and how
this is done the transformation language.
Chapter 4

Transformation Engine Design

Endless Loop: n., see Loop, Endless.
Loop, Endless: n., see Endless Loop.

– Random Shack Data Processing Dictionary

The aim of the transformation engine is to take the transformation database and apply a set of applicable transformations to the input code. There are two facets to the design of the transformation engine: how the engine itself is structured and how transformations should be applied.

The application of transformations is applicable regardless of how the engine is structured. This is covered first, in Section 4.1.

Section 4.2 discusses the design of the prototype implementation used for testing.
4.1 Making Transformation Decisions

This section discusses the factors which need to be considered when applying transformations to a source tree. Each of the three main factors discusses the problem to be considered, then presents the solution adopted for this project.

4.1.1 Order of Application

Discussion of the Problem

It is desirable to achieve the best possible result from a given set of transformations, by applying them in the correct combination.

In hardware designs, the optimisations which give the best result are not necessarily those that would appear to at first glance. Expanding a loop with no loop-carried dependencies to a parallel execution will certainly decrease the number of cycles taken. However, the extra hardware resources required may lead to path congestion or running out of resources completely. In this situation, the longest path through the resulting circuit may drop the $f_{\text{max}}$ so far that the overall execution time rises.

The most effective way to determine which combination of optimisations produces the best results is to try all of them. This trial-and-error method is known as *design space exploration* and is a time-consuming process. Even on basic designs, a single iteration can take upwards of 10 minutes on a modern workstation. Complex designs can take hours to go from high-level source to low-level hardware design. Clearly this makes an exhaustive check of all combinations impractical except in the very final stages of design, or with a highly parallel processing capability.

Distributed design-space exploration utilising multiple machines can provide fast results if the problem can be divided up in a sensible manner. The user may then be presented with results to select the variation which provides the best behaviour for their requirements. Figure 4.1 shows the theoretical design flow for this option.

![Figure 4.1: Design Space Exploration — Theoretical Design Flow](image)

A possible solution to this problem is to use modelling techniques instead [32, 38]. Models

---

1What constitutes “best” is dependent on the circumstances. Whilst increased execution speed may be desirable in some cases, the potential corresponding increase in area may be the overriding factor if hardware resources are limited.

38
can determine the effectiveness of transformations without passing each variation through the full tool chain. Whilst not as accurate as the exhaustive search, this method could produce good results in a fraction of the time. This technique is strongly dependent on having reliable and accurate models.

One major drawback to the modelling approach is adequately representing the interactions between all of the components involved in determining performance characteristics. Extensive models must be held for all targets, requiring them to be updated when new targets are introduced (the introduction of a new FPGA for example). Modelling also does not reduce the overall search space, as due to the local maxima problem as shown in Figure 4.2 all results must still be generated for modelling. To summarise, using modelling in this context:

- Is inaccurate,
- Increases the need for maintenance,
- Does not decrease the search space.

Proposed Solution

The solution presented here generates an entire search space of designs, by creating a new solution when it applies a transformation. The generated solutions can be passed on for further processing, which could include:

- A separate pass for estimation of the best solution using models.
- Synthesis into a hardware design on the target platform, providing real performance data.

The major problem with design-space exploration is the size of the result-set involved. Each transformation applied doubles the number of total solutions, which on big designs has the potential to generate an enormous exploration space. Figure 4.3 illustrates one of the decisions taken to mitigate this problem. The number of duplicated parts of the tree is kept to a minimum as long as possible to reduce memory usage and execution time. A further attempt to reduce this search space is the inclusion of “always” transformations — those which are always applied and do not generate two solutions in the search space. This is most often used for transformations which alter the syntax only, but not the control or data flow. This must be used sparingly, typically only when it is known that the transformation has a positive effect (or no effect if it allows other transformations to be applied — see Section 4.1.2 for why this is needed).

As described in Section 2.2.3 the typical design process only allows the large amount of time required for design space exploration in the closing stages. During initial development, a faster solution is desirable. For this reason, the transformation engine has a single-solution mode, where it will attempt to apply many transformations as possible to the input. There is scope within this to add heuristics to determine the best transformations to apply, whilst retaining one solution at all points. This report discusses some possible heuristics in Section 7.4.5.

\footnote{Due to the length of time it takes, repeating it for each iteration of a design is wasteful.}
After hitting a local maxima,
subsequent optimisations 
DECREASE performance. 
Search would stop here.

Higher performance is 
possible by continuing to 
apply transformations, 
rather than giving up

Figure 4.2: This simplified diagram illustrates the Local Maxima problem. Along the x-axis are different code versions generated by the transformation engine, in the order which they are generated. A basic algorithm may apply transformations until it detects a decrease in the estimated performance, at which point it would give up. As can be seen, continuing to generate solutions would yield a far better result.

Figure 4.3: In design space exploration mode, the transformation engine generates multiple options by branching whenever a transformation is applied. To keep memory usage small for as long as possible, this branch point is kept in place whilst further optimisations are applied. At the end of a transformation pass, the option points are propagated upwards to form two distinct solutions. New matches may be found by further passes once the branch point is removed.
4.1.2 Composite Transformations

Discussion of the Problem

When combined, transformations have the potential to act in several ways. There is the possibility that two transformations may combine to produce a greater increase than either on its own (the ideal solution). Unfortunately, there is also the potential for optimisations to reduce performance when applied together. The use of design-space exploration techniques should remove the case where combined transformations cause performance to drop. With this in mind, it is preferable for the transformation engine to attempt to compose as many transformations as possible.

To support composite transformations, the transformation engine needed to:

- Automatically compose transformations to produce better results,
- Attempt to restructure the program to increase the applicability of user-defined transformations.

Proposed Solution

The need for automatic composition led to the adoption of a multi-pass transformation system, where transformations are applied repeatedly to the tree.

Restructuring to enable additional transformations is a little more complicated, however some basic restructuring can provide good improvements. The approach adopted here is to perform restructuring using programmable transformations, defined in the same way as any other transformation in the system. Examples of this include:

- Converting nested parallel blocks into a single block,
- Converting nested sequential blocks into a single block,
- Converting for loops to while loops,
- Converting for loops with only one statement in the loop body to for loops with a sequential block containing one statement (so all for loop transformations can be defined with a sequential block).

These restructuring transformations can be found in Section A.1.

4.1.3 Proof of Correctness

Discussion of the Problem

Ensuring a transformation preserves data integrity when applied is an important factor. A formal proof of the correctness of a transformation (including all integrity conditions) and the tool applying it is required to demonstrate true correctness of an optimisation. In a programmable transformation engine, it is undesirable to undertake a formal verification whenever the user adds a new transformation to the database.
Figure 4.4: Automatic Parallelisation of a While Loop with Composite Transformations

An alternative approach is to verify correctness after the application of a transformation [39]. This provides a concrete optimisation to work with, however should the optimisation be proven invalid it would have to be reversed. Unfortunately, this added verification step after every transformation would reduce performance so it is perhaps only suitable for user-added transformations which have not been formally verified.

Proposed Solution

The approach taken here is to informally argue for the validity of the transformations presented. It is envisaged that a formal validation step could be added as an additional pass within the transformation engine. Validity arguments are presented in Chapter 5.
4.2 Transformation Engine Architecture

The main possibilities for the transformation engine were:

- Build on Todman’s Cobble compiler, adding an additional pass.
- Take parts such as the parser from Cobble and build on those alone.
- Implement a completely new system.

With key compiler parts such as a parser and AST definition already present in Cobble, the first two options were the most sensible. A dedicated source-to-source transformation engine using key parts of Cobble was possible, however adding an additional pass to Cobble seemed to be the best solution, giving the potential for a direct hardware output. An added advantage was a reduced risk of spending a large amount of time chasing problems caused by removing code from the full Cobble environment.

4.2.1 Integrating into the Compiler

Choosing Cobble as the base for the transformation engine introduced several additional choices for integration.

The options, with arguments for and against are presented below and illustrated in Figure 4.5.

![Diagram of options for integration with Cobble.](image)

**Figure 4.5:** Options for Integration with Cobble. Part (a) illustrates the to CML to C++ approach, with transformations loaded as dynamic link libraries (Windows) or shared objects (Linux). Part (b) shows the chosen option, an integrated compiler pass.
Option 1. Translate the CML into C++, then compile into a DLL or shared object. Dynamically load transformations via a plug-in architecture.

The C++ approach provides high-performance for each single transformation, as it can be fully optimised by the C++ compiler. The plugin architecture allows them to be dynamically loaded at runtime, so the user need not recompile the whole compiler.

Having established that a design-space exploration mode was needed, any plugin architecture begins to look undesirable. Each transformation must be passed a description of the AST, modify it, then pass it back. Passing direct pointers to the AST in memory within the Cobble compiler would result in a dependency on the internal AST format, so a plugin solution would require data copying and format conversion. On a single Cobble design this is an acceptable solution, but when many hundreds of solutions are being generated the impact on speed and memory requirements would be unacceptable.

Isolating transformations in this manner reduces the ability of the transformation system to see the bigger picture, reducing its ability to make informed decisions about which transformations to apply — each transformation would need to decide based on its own view, removing the possibility of an overall optimisation strategy.

Requiring a C++ compiler to build a new library every time a new transformation is created would work well for generic transformations where most would be defined in advance, however for application-specific transformations this would be a burden on the end user.

Option 2. Add an integrated compiler pass which parses the CML into an AST, then match with pattern matching.

This approach uses a single transformation engine instance with a knowledge of every transformation in the database. Using the visitor pattern support in Cobble, a tree walker can walk the internal Cobble AST and modify it as transformations are applied. Rather than one walk of the AST per transformation, per pass, the AST need only be searched once per pass.

Keeping the transformation engine inside the compiler reduces the need for data copying and format conversion.

4.2.2 Cobble-CML Architecture

Producing a full plug-in framework for Cobble to facilitate the addition of transformations would have provided a powerful ability to dynamically load compiler passes. However, it introduces extra problems of data marshalling across the plug-in boundary. With a limited amount of time and a desire to explore the behaviour of transformations defined in the manner described here, an integrated pass was chosen as the approach more likely to achieve results.

Figure 4.6 shows a basic overview of the system, with the transformation engine operating directly on the Cobble AST before hardware generation.

Figure 4.7 illustrates the additional passes added to the Cobble compiler. After a design has been parsed and type-checked, the first new compiler pass performs the actual transformation process, using tree matching as described in Figure 2.2. A visitor-pattern tree walker descends the tree node-by-node, applying transformations that match at each one.

The process of transformation may produce an AST with branch points, as seen in Figure 4.3. After propagation, an AST with a single set of complete solutions is produced. Each solution
is type-checked again as a partial check of correctness and to allow types to be assigned to any new variables generated in the transformation process, as described in Section 3.2.4. Types of new variables are inferred from their context during type-checking.

The final pass is an output stage, which generates Handel-C or Cobble source from the AST description. The original Pebble output stage is retained, allowing the compiler to generate a hardware design when only one solution is generated (i.e. the compiler is not in design-space exploration mode).

![Figure 4.6: Final Solution Overview — Transformation Engine Operates on AST](image)

**Figure 4.6:** Final Solution Overview — Transformation Engine Operates on AST

![Figure 4.7: Cobble-CML Architecture: Additional Compiler Passes. The additions to the system are represented by orange boxes.](image)

**Figure 4.7:** Cobble-CML Architecture: Additional Compiler Passes. The additions to the system are represented by orange boxes.
4.2.3 Changes to the Cobble AST

Adding support for transformations required changes to the existing Cobble AST and some new additions to it. Originally, it was not possible to restructure the AST without copying it, restructuring during the copy. For efficiency, the AST was modified to permit on-the-fly restructuring during the transformation process.

The transformation language introduces new features such as generic variables, expressions statements and statement lists. These were added as an extension to the existing Cobble AST as simple subclasses of existing variables, expressions and statements. The extended AST is only used when loading CML transformation definitions, as the original Cobble language is unchanged.

Cobble supports the visitor pattern throughout, so an extended set of visitor classes were produced to support the new AST additions.

4.2.4 Compatibility Issues

The output format produced by the original Cobble compiler was Pebble 3.0 [40, 41]. This was a decision made at the start of development work on Cobble, however since that time Pebble has moved on to version 5.0. The two versions are not syntactically compatible and working version 3.0 compilers are hard to find. The Pebble 5.0 compiler is a much more robust, co-ordinated effort but has a modified syntax.

A converter from Pebble 3.0 to Pebble 5.0 is available, but this is a Java program which does not successfully convert all Pebble 3.0 files.

With both a Pebble 3.0 compiler and a 3.0 to 5.0 converter which did not work, the decision was taken to modify Cobble to output Pebble 5.0.

In addition to the Pebble output for hardware descriptions, a Handel-C output pass was added to allow translation from Cobble to Handel-C, or the generation of multiple Handel-C files for the multiple solutions produced in design-space exploration mode.

4.2.5 Compiler Toolchain

The final Cobble-CML compiler toolchain, showing both Handel-C and Pebble routes, is shown in Figure 4.8.

![Figure 4.8: The Cobble-CML Toolchain](image)

The route via Celoxica’s DK Design Suite is faster and more mature, so proved more suitable for testing and results.

46
4.3 Summary

This chapter has discussed the issues involved when designing the transformation engine. It presented the Cobble-CML compiler, which has the following operating modes:

- **Direct hardware compilation** - no transformations are applied.
- **Straight transformation** - as many transformations as possible are applied, in the order in which they are defined.
- **Design-space generation** - an entire design space of solutions is generated, with different combinations of optimisations.

The following changes to the Cobble framework were described:

1. Addition of three new compiler passes (transformation, post-transformation type-checking and Handel-C output),
2. Addition of CML features to the Cobble AST,
3. Modification of the Cobble AST for increased efficiency during transformations,
4. Conversion of output from Pebble 3 to Pebble 5.
Chapter 5

Transformations

This chapter presents a variety of transformations created using the transformation engine. For brevity in the main body of this report, source listings and validation arguments for a wider range of transformations are available in Appendix A.

The transformations described here are:

- **Generic**
  - Automatic Parallelisation,
  - For To While Loop Conversion,
  - Converting < to ==.

- **Application Specific**
  - Parallel Matrix Multiply,
  - Matrix Shift Load.

Appendix A covers the following additional transformations:

- Restructuring transformations from the standard library,
- Automatic parallelisation of for loops with no loop-carried dependencies,
- Factorisation of arithmetic expressions to reduce multipliers,
- Dead code removal,
- Replacement of variables in expressions by the expression last assigned to them, to enable parallelisation.

The transformations in this chapter were all used during testing for Chapter 6.
5.1 Generic Transformations

5.1.1 Automatic Parallelisation

An effective transformation for C-based code is automatic statement parallelisation. A lot of C-code contains straight blocks of initialisation and re-initialisation of unrelated variables.

With two consecutive statements, if the second is not waiting on the first and they do not assign to the same place then there is no conflict and they may be executed in parallel.

Further improvements can be made under Handel-C: if the first statement uses a variable assigned to by the second statement, Handel-C’s timing ensures that the assignment occurs after the first statement has read the value, even when executed in parallel.

The auto_par transformation shown below is the CML implementation of this:

```cml
transform auto_par {
  pattern
  {
    cmlstmtlist(preamble); // statements before
    cmlstmt(par1);        // first statement
    cmlstmt(par2);        // second statement
    cmlstmtlist(postamble); // statements after
  }
generate
  {
    cmlstmtlist(preamble);
    par
    {
      cmlstmt(par1);
      cmlstmt(par2);
    }
    cmlstmtlist(postamble);
  }
  conditions
  {
    // don’t assign to the same place
    defs(cmlstmt(par1)); & defs(cmlstmt(par2)); == {};

    // second statement not waiting on first
    defs(cmlstmt(par1)); & uses(cmlstmt(par2)); == {};
  }
}
```
5.1.2 For To While Conversion

Celoxica provides a document titled “Implementing Efficient Loops in Handel-C” [4]. This recommends converting a for loop to a while loop, then placing the loop increment statement in parallel with the last statement of the loop body, as discussed in Section 2.9.1.

Parallelisation is already covered by the previous transformation, so the simple definition below is used as an intermediate transformation. It only exists to allow other transformations to have an effect.

The match and substitution are identical semantically and generate the same data flow and control flow graphs.

```plaintext
transform for.to.while {
    pattern
    {
        for(cmlstmt(init); cmlexpr(cond); cmlstmt(step))
        {
            cmlstmtlist(body);
        }
    }
    generate
    {
        cmlstmt(init);
        while(cmlexpr(cond))
        {
            cmlstmtlist(body);
            cmlstmt(step);
        }
    }
}
```
5.1.3 Converting \texttt{<} to \texttt{==}

Many programmers structure a for loop iterating from \texttt{a} to \texttt{n} as shown in the pattern below — with the \texttt{<} operator to check the upper bound. However, in hardware an equality check should be preferable. This transformation verifies that the replacement of \texttt{<} with \texttt{==} is valid and transforms the loop. Clearly this must be applied \textit{before} a for loop is converted to a while loop.

The conditions section ensures that the loop variable is only modified within the iteration and increment parts of the loop, with it counting upwards in a fixed step which will eventually reach the maximum value defined (and not skip over it).

```
transform \texttt{lt\_to\_eq} {
  cmlvar i;
  pattern {
    for(i = cmlexpr(init); i < cmlexpr(comp); i += cmlexpr(inr))
    {
      cmlstmtlist(body);
    }
  }
  generate {
    for(i = cmlexpr(init); i != cmlexpr(comp); i += cmlexpr(inr))
    {
      cmlstmtlist(body);
    }
  }
  conditions {
    constant(cmlexpr(init)); // constant initialise
    constant(cmlexpr(comp)); // bound is not a variable
    constant(cmlexpr(inr)); // iterator is incremented by a constant

    // lower bound is less than upper bound
    eval(cmlexpr(init) < cmlexpr(comp));
    // count upwards
    eval(cmlexpr(inr) > 0);
    // it will reach the bound exactly
    eval(((cmlexpr(comp)−cmlexpr(init)) % cmlexpr(inr) == 0);
    // must not change i in the loop!
    defs(cmlstmtlist(body)); & \{i\} == \{\};
  }
}
```

52
5.2 Application Specific Transformations

5.2.1 Parallel Matrix Multiply

This is an application-specific transformation, which executes the inner loop of a matrix multiplication in parallel. It also demonstrates the creation of new variables.

This is an example of parallelism through hardware duplication, so is expected to reduce the cycle count but increase the resource usage.

Variables t0, t1 and t2 are created when the matched part is replaced.

```plaintext
always transform matmult {
  cmlvar i, j, k, aa[[]], bb[[]], cc[[]], t0, t1, t2;
  pattern {
    for (i = 0; i < 3; i++)
      {
        for (j = 0; j < 3; j++)
          {
            for (k = 0; k < 3; k++)
              {
                cc[i][j] = cc[i][j] + aa[i][k] * bb[k][j];
              }
          }
    }
  }
  generate {
    for (i = 0; i < 3; i++)
      {
        for (j = 0; j < 3; j++)
          {
            par
              {
                t0 = aa[i][0] * bb[0][j];
                t1 = aa[i][1] * bb[1][j];
                t2 = aa[i][2] * bb[2][j];
              }
            cc[i][j] = t0 + t1 + t2;
          }
      }
  }
}
```
5.2.2 Matrix Shift Load

This is an application-specific transformation, which converts a sequential indexed array load into a shifted load as described in Section 2.9.2.

```cml
transform matrix_shiftload {
  cmlvar loadi, In1, In2, loadBufA [], loadBufB [];
  /* This pattern matches directly to the source */
  pattern {
    while(loadi != cmlexpr(max))
    {
      par
      {
        loadi ++;
        loadBufA[loadi] = In1;
        loadBufB[loadi] = In2;
      }
    }
  }
  generate {
    /* Replace it with something more complicated .. but the original * source remains intact. */
    while(loadi != cmlexpr(max))
    {
      par
      {
        loadi ++;
        loadBufA[cmlexpr(max)-1] = In1;
        loadBufB[cmlexpr(max)-1] = In2;

        par(q = 0; q < cmlexpr(max)-1; q++)
        {
          loadBufA[q <- 4] = loadBufA[(q+1) <- 4];
          loadBufB[q <- 4] = loadBufB[(q+1) <- 4];
        }
      }
    }
  }
  conditions {
    constant(cmlexpr(max));
  }
}
```

5.3 Summary

This chapter has presented a range of sample transformations and their descriptions in the variation of CML defined in this report. Both generic and application-specific ones have been demonstrated, with varying degrees of data-integrity conditions and complexity.
Chapter 6

Testing and Results

Guard against the prestige of great names; see that your judgments are your own; and do not shrink from disagreement; no trusting without testing.

– Lord Acton Dalberg

This chapter presents quantitative results from tests using manageable examples and discusses the implications for the optimisation mechanism described here.

A single example is used for a variety of tests and the results analysed, with additional examples covered in lesser detail to support the findings.

Section 6.1 covers the testing process used.

Section 6.2 presents the results of testing with multiple transformations applying to the same base design, with an analysis of the results.

Section 6.3 provides less detailed results for additional examples to support the findings of the major example. It provides supporting evidence that the transformations described as generic in Section 5.1 produce an improvement in other scenarios.
6.1 Test Setup

6.1.1 Major Test Case

The primary example used for testing is a simple 3x3 matrix multiplication. Although simple, it has the following important features:

- Reflects the algorithm directly.
- Nested loops.
- Multiplication.
- Sequential statements with no dependencies.
- Illustrates many important points transferable to more complex examples.

The example code is passed through the transformation engine, to generate several possible solutions. Each version is compiled to EDIF by Celoxica’s DK3.1 compiler and then passed to Altera’s Quartus II 5.0 or Xilinx’s ISE 7.1.

Four distinct targets were used for testing the system, two from both of the main FPGA vendors:

- **Altera Stratix EP1S10F7806ES** - One of the smaller FPGAs in the Stratix family, which contains specialised DSP blocks for multiply and multiply-accumulate operations.

- **Altera Cyclone EP1C12F256C6** - A chip with a similar number of logic elements and similar architecture as the Stratix above, however without the DSP blocks.

- **Xilinx Virtex II XC2V1500-6BG575** - The Xilinx counterpart to the Stratix in this test, the Virtex contains dedicated hardware multiplication blocks.

- **Xilinx Spartan 3 XC3S1500-5FG676** - The Xilinx counterpart to the Cyclone in this test.

The base design for the 3x3 matrix multiplier uses approximately the same percentage of logic cells and has approximately the same $f_{max}$ on both the Stratix and Cyclone. The Xilinx parts were chosen so area used for the base design is approximately the same as that on the corresponding Altera part, with the $f_{max}$ as close as possible.

Handel-C’s technology mapping was disabled during the core set of tests, to prevent mapping of multipliers to dedicated multiplier blocks. The tests for the Stratix were then re-run with technology mapping turned on, allowing DSP blocks to be allocated. These results are marked as Stratix DSP. In the interests of fairness, the same was tried with the Virtex II, however the Handel-C compiler failed to map to any multiplier blocks.

Throughout the tests, there are a set of standard transformations which are always applied, to enable further progress. These are described in Section A.1.

Source for the original source and some of the transformed versions can be found in Section C.1.
6.1.2 Supporting Test Cases

There are three supporting test cases, which all use an Altera Stratix EP1S40C5, for which a test board was available.

The first supporting case is a variation of the matrix multiply experiment, with an infinite multiplication loop instead of the one-shot variation used in the major test. This case is used to retrieve a dynamic power estimate.

The two remaining supporting cases are simple kernels of a Sobel edge detection filter and AES table generation. They are passed through the transformation engine with the four generic transformations enabled, with results presented in a set of tables in Section 6.3.
6.2 Effect of Multiple Transformations

This set of tests examines the effect of applying several transformations to a design.

The transformations presented in Chapter 5 are used:

- **autopar** - Automatic parallelisation of adjacent instructions with no dependencies.
- **fortowhile** - Conversion of while loops to for loops to expose more parallelisation opportunities.
- **lttoeq** - Replace < with == in incrementing for loops.
- **matrixpar** - Design-specific transformation to parallelise the central multiply-accumulate loop.

Starting from the base design, each transformation is added to those applied until they are all applied together.

Where “best execution time” is used, this refers to the execution time if the design was run at $f_{max}$ for the target platform.

### 6.2.1 Results

Table 6.1 illustrates the effectiveness of the automatic parallelisation features on the cycle count of a design. Simple parallelisation of consecutive but non-dependent statements yields almost a 20% reduction in the number of cycles. Addition of the for-to-while loop transformation makes it a 45% reduction. Finally, the application-specific matrixpar transformation provides a 58% reduction in number of cycles, at the expense of duplicated hardware.

<table>
<thead>
<tr>
<th>Version</th>
<th>base</th>
<th>autopar</th>
<th>fortowhile</th>
<th>lttoeq</th>
<th>matrixpar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>188</td>
<td>152</td>
<td>104</td>
<td>104</td>
<td>80</td>
</tr>
</tbody>
</table>

**Table 6.1: Number of Cycles for Matrix Multiply**

Table 6.2 illustrates the logic usage for each design on the platforms tested. Xilinx and Altera logic element measurements are not comparable, so the trend is of importance here. Note that with the transformations which enable parallelisation of sequential statements (autopar and fortowhile), the logic usage drops across all platforms.

**Conclusion:** Simple parallelisation of statements does not increase resource usage. Rather, it reduces it slightly.

Figures 6.1 and 6.2 combine the best execution time data and logic usage, providing the most illustrative set of results presented here. The graphs have two regions — a vertical stack of points and an isolated point.

### 6.2.2 Illustration of “Inefficient” Code

The vertical stacks in Figures 6.1 and 6.2 illustrate “inefficient” code. All variations in this stack have approximately the same area usage, however the top three all take longer to
Figure 6.1: Matrix Multiply Area/Execution Time Comparison for Stratix
Figure 6.2: Matrix Multiply Area/Execution Time Comparison for Virtex
execute. Since it is possible to take 45% off of the execution time and also save a small amount of area, why use the base version at all? The answer of course is power.

Figure 6.3, from the power estimation supporting test, shows the estimated change in power with execution time for the different code versions at 50MHz. With a fixed clock frequency any gains in execution time can only be made by reducing the number of cycles taken, which implies increased parallelism. This is illustrated here, where decreased execution time leads to a higher power requirement as more statements are executed in parallel.

**Conclusion:** There is scope with naive code to introduce large performance improvements, with no cost in terms of area. However, these improvements introduce a power cost.

**Conclusion:** Parallelisation via hardware duplication (as in matrixpar) can produce large performance improvements, however with an expected increase in area.

### 6.2.3 Illustration of Composite Transformations

The *autopar* and *fortowhile* transformations form a basic but illustrative example of composite transformations.

The major advantage of smaller transformations is when verifying them as valid — formally or informally. The informal validity argument for *for-to-while* is simpler than it would be if it was to parallelise at the same time, a function the automatic parallelisation can perform in any case.

This becomes particularly useful if enabling transformations can be defined — that is, those which are only applied when they enable other transformations to occur. The for-to-while transformation is one of these, as it serves no purpose other than to allow future parallelisation to take place.

**Conclusion:** A combination of simple transformations may have the same effect as a single, more complex transformation. This is preferred, as validity is easier to verify.
Figure 6.3: Dynamic Power Estimate and Execution Time
The Need for Design Space Exploration

Figures 6.1 and 6.2, when taken together, show why identifying the best transformations to apply is a difficult task and justifies the need for a design-space exploration mode, as discussed in Section 4.1.1.

Of the designs with similar resource usage, `lttoeq` is the fastest on the Stratix, whilst `fortowhile` is the fastest on the Virtex.

6.2.4 Parallelisation of Statements vs Hardware Duplication

An important point can be made here regarding the use of parallelisation. There are two distinct parallelisation transformations in this set of tests:

- Consecutive Statement Parallelisation (`autopar`) - Pre-existing statements are parallelised, allowing them to run concurrently rather than sequentially.
- Hardware Duplication (`matrixpar`) - A single hardware block is replicated several times, to perform disjoint operations in parallel.

Unless power is an issue\(^1\), parallelisation of existing hardware is essentially free as can be seen from the `matrixpar` and `fortowhile` tests.

On the other hand, as expected, duplicating hardware to run in parallel costs not only in terms of power but resource usage too. The increase in \(f_{\text{max}}\) of the `matrixpar` test is likely to be due to a reduction of the longest path in the design by the elimination of a `+=` operator.

6.2.5 Raw Performance

With large FPGAs and non-mobile devices, it is acceptable to consider performance preferable to conserving area or reducing power. Figure 6.4 shows the progressive change in best execution time as each new transformation is applied.

As can be seen, there is a general and substantial downward trend. Figure 6.5 shows that with the exception of `matrixpar`, most of this downward trend is due to the decrease in cycles. On the Altera devices and the Spartan 3 automatic parallelisation has only a small effect on \(f_{\text{max}}\), although it is slightly greatered on the Virtex II.

\(^1\)Of course, power is becoming much more of an issue with the move to mobile devices
Figure 6.4: Matrix Multiply Best Execution Time.
Figure 6.5: Matrix Multiply $f_{max}$ Comparison
6.2.6 Architectural Considerations

Figure 6.4 illustrates the effect of architectural differences on the effectiveness of transformations. There are some interesting points to note:

Observation:
- The effect of transformations on Altera chips was consistent.
- With DSP mapping turned off, the Cyclone was faster than the Stratix.

Possible Explanation: Altera’s Stratix and Cyclone FPGAs have a similar architecture, however the Cyclone has less on-board memory and no DSP blocks. Routing around these specialised blocks is not required in the Cyclone, making routing simpler and potentially leading to the increased performance. The good news for Altera is that the more expensive Stratix beats the Cyclone when the DSP blocks are brought into use.

Observation: Converting a < comparison to an == comparison had little effect on Altera chips, but decreased \( f_{\text{max}} \) on both Xilinx chips.

Possible Explanation: Looking at the differences between the Virtex II and Stratix, performance is closest just before the \( \text{ltoeq} \) transformation is applied. This may indicate that the Xilinx tools or architecture (perhaps the fast carry chains) deal effectively with a < comparison, an advantage which is lost when removing it.

Table 6.2 shows that the \( \text{ltoeq} \) transformation results in more slices being used, so resources are being used less efficiently after the transformation.

This also demonstrates effectively that transformations which intuitively should result in a performance improvement have the opposite effect.

Observation:
- The Spartan 3 was not consistent with the Virtex II.
- The Spartan 3 suffered appalling performance initially, however caught up as more optimisations were applied.
- The Virtex II closed the gap on the Altera parts as more optimisations were applied.

Possible Explanation: The consistency difference between Spartan 3 and Virtex II results from them being two different generations of chip, with Spartan 3 being the newer architecture. The performance improvement on the Spartan 3 was dramatic, as the decrease in cycle count has a greater effect at lower clock rates.
6.3 Results On Other Programs

The Matrix Multiply example has been covered in great detail, as it illustrates many key points. The reader may, however, not be convinced that the transformations are applicable to a variety of programs. Tables 6.4 and 6.5 illustrate results achieved on other programs for Altera’s Stratix EP1S40C5 part. There is no program-specific optimisation, so the comparison is with the lttoeq configuration. A decrease in cycle count is seen thanks to automatic parallelisation, with the corresponding small reduction in resource use. The measures of logic usage are Altera Stratix logic elements through these additional tables.

The programs used are:

- **matmultinf** - an infinitely looping matrix multiply — the number of cycles is for one iteration.
- **sobel** - the core of a Sobel transform on a 10 pixel x 10 pixel image.
- **aes** - a kernel taken from AES.

<table>
<thead>
<tr>
<th>Program</th>
<th>matmultinf</th>
<th>sobel</th>
<th>aes</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>189</td>
<td>4686</td>
<td>1948</td>
</tr>
<tr>
<td>lttoeq</td>
<td>106</td>
<td>2818</td>
<td>1022</td>
</tr>
<tr>
<td>% Decrease</td>
<td>44%</td>
<td>40%</td>
<td>48%</td>
</tr>
</tbody>
</table>

Table 6.4: Number of Cycles After Transformation for Other Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>matmultinf</th>
<th>sobel</th>
<th>aes</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>3076</td>
<td>4606</td>
<td>6850</td>
</tr>
<tr>
<td>lttoeq</td>
<td>3057</td>
<td>4593</td>
<td>6839</td>
</tr>
</tbody>
</table>

Table 6.5: Logic Usage After Transformation for Other Programs

Table 6.6 shows the full set of results for the Sobel filter supporting test. A similar trend can be seen to the main Matrix Multiply test, with automatic parallelisation having an effect initially, but being aided greatly when for loops are converted to while loops. The trend with the lttoeq transformation also reappears — this test is on an Altera part and a small improvement in \( f_{max} \) is seen.

<table>
<thead>
<tr>
<th>Platform</th>
<th>base</th>
<th>autopar</th>
<th>fortowhile</th>
<th>lttoeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Usage</td>
<td>4606</td>
<td>4603</td>
<td>4600</td>
<td>4593</td>
</tr>
<tr>
<td>Cycles</td>
<td>4686</td>
<td>4285</td>
<td>2830</td>
<td>2830</td>
</tr>
<tr>
<td>( f_{max} )</td>
<td>84.98</td>
<td>84.13</td>
<td>82.02</td>
<td>85.97</td>
</tr>
<tr>
<td>Best Execution Time (( \mu s ))</td>
<td>55.14</td>
<td>50.93</td>
<td>34.50</td>
<td>32.92</td>
</tr>
</tbody>
</table>

Table 6.6: Execution Time and Logic Usage Results for the Sobel Supporting Test
6.4 Summary

This chapter has illustrated the success of the programmable transformation engine approach, with performance improvements in excess of 70% in some cases. It has covered the interesting patterns which emerged through testing on several platforms and suggested possible explanations.

Testing revealed that:

- Automatic parallelisation is possible with the transformation engine, with good results,
- There is much room for improvement with C-style Handel-C code, but some of this can be achieved automatically,
- Transformations may behave differently on different hardware platforms.
Chapter 7

Conclusion

*I haven’t failed, I’ve found 10,000 ways that don’t work.*

– *Thomas Edison*

This project started with the following primary aims:

1. Design a user-programmable transformation system for Handel-C,
2. Produce transformations to achieve the same improvements as an experienced Handel-C designer.

Secondary aims were to use the programmable transformation system to:

1. Gain an insight into the way transformations interact,
2. Understand how transformation results differ between hardware platforms.

Section 7.1 covers the degree of success to which the aims were met, while Section 7.2 discusses the current limitations.

Section 7.3 describes how the transformation engine can be used to provide answers to the real questions being asked about designs.

Section 7.4 put forward ideas for enhancements to the transformation system to increase the diversity and applicability of the transformations that can be described.

Section 7.5 explores possible avenues applications for the transformation engine beyond optimisation.
7.1 Meeting the Aims

The aims for this project were divided into two parts, as the secondary aims relied on at least partly achieving the primary ones.

The first aim of this project was to create a user-programmable transformation system for Handel-C. This report defines a transformation language and transformation engine, which together form that system. As it stands it works well, generating one or more different solutions as transformations are applied.

Only a prototype has been developed, so there is scope for many more additions to the language and transformation engine, which are covered in Section 7.4. One major limitation at present is the lack of support for the optimisation of macro procedures.

Speed and memory usage is currently a problem, due to a number of factors:

1. Basing the solution on Cobble leads to a lot of extra items in the AST which are not required for pure transformations,
2. The algorithm used to search for matches is a brute force one,
3. The matching algorithm cannot operate over branch points in design space exploration mode.

These can be solved in the following ways:

1. Separate the transformation engine from Cobble, or allow it to use a cut-down AST,
2. Improve the search algorithm to avoid repeated descent of the tree,
3. Improve the matching algorithm so propagation of branches is not required, to reduce memory usage.

The second aim was to try and produce results similar to what an experienced Handel-C designer could achieve. Due to the limitations of the transformation language, a more restricted set of transformations than desired can be described, so this can not be completely met. The additions discussed in Section 7.4 would permit a wider range of transformations and bring the transformation language closer to that aim.

The transformations that have been successfully described make solid progress, delivering large performance improvements and reducing inefficient hardware. This is closer to a medium-level Handel-C designer, however would certainly be a time saver. It is of particular use to those new to Handel-C, as they are most likely to write the style of code which benefits most from the transformations defined for testing in Chapter 6.

The secondary aims required a working transformation engine to generate results to analyse. Thankfully the transformation system is competent at this task, delivering a variety of optimised solutions which deliver some interesting results.

The two secondary aims are tightly linked, both involving the interaction of a transformation with its environment (be it other transformations applied before and after it, or the final hardware target). Even though a small example was used to ensure results could be collected in a short space of time, some interesting patterns were identified in Chapter 6.
Analysis of the results revealed that the performance of differing architectures converged as the level of statement-level parallelism increased, as the lower clock rates of the slower ones became less important as the number of cycles required reduced. It was demonstrated that transformations that should improve performance will not always do so and whether there is a performance improvement or not can depend on the target architecture. This result justified the presence of a design-space exploration mode.

7.2 Limitations

After some experience with the transformation system it became apparent that some of the transformations originally considered, such as converting sequential array accesses to shifts (Section 2.9.2) would be difficult or impossible to define generically in the transformation language, even with additional language extensions 7.4.1. It is possible to define for a limited set of circumstances, such as single statement loops with simple operators, sfor instance a vector addition or matrix multiplication operation.

Much time and effort was spent working and adapting Cobble to add the transformation engine, with the hope of making it generate multiple hardware designs automatically, without the intermediate source stage. Unfortunately the design of Cobble was not amenable to this and the Pebble output method struggled with performance on large designs. In hindsight, this time would have been better spent implementing some of the additional language extensions mentioned in Section 7.4.

7.3 Solving the Real Problems

The examples presented here are deliberately small, to make them manageable for analysis in the time available. They demonstrate that the solution presented here shows promise.

The real question is not whether this solution meets the goals set out so far, but if it could be further adapted to meet other real problems being faced at the moment. These problems are answers to the questions:

- What is the fastest design using $x\%$ of the chip’s resources?
- What is the smallest design executing in $x$ seconds?
- What design offers the best tradeoff between space, size and power?
- Would the extra cost of a larger chip be worth the increase in performance?

The design-space exploration feature of the transformation engine allows these questions to be answered. As discussed previously (Section 4.1.1), the most accurate results for these variables are found by performing a full place-and-route on each design. Models could be used as a less accurate alternative.

Regardless of the method used, once results for time, resource usage and power are collected all of the questions above can be answered using careful analysis.
7.4 Future Work

The experience of building the prototype transformation engine and example transformations demonstrated many avenues for future work, some to resolve unforeseen circumstances in the original design and some for enhanced abilities.

7.4.1 Transformation Language Extensions

Handel-C — and by extension CML — supports parallel and sequential replicators. These operate at a statement level, but there is not a comparable feature at expression level. The transformation language would benefit greatly from an enhanced matching and replacement engine with a notion of iteration on expressions.

The `par` construct expands as follows:

```
par( i = 0 ; i < 3 ; i++) {
  a[ i ] = b[ i ] * c[ i ];
}
```

becomes

```
par {
  a[ 0 ] = b[ 0 ] * c[ 0 ];
  a[ 1 ] = b[ 1 ] * c[ 1 ];
}
```

A similar construct for expressions could be:

```
a = expr(+; i = 0; i < 3; i++; b[ i ] * c[ i ]);;
```

becomes

```
a = (b[ 0 ] * c[ 0 ]) + (b[ 1 ] * c[ 1 ]) + (b[ 2 ] * c[ 2 ]);;
```

The expression is replicated, with each replication being joined by the operator specified first.

This concept could also be extended to matching. For example, it would be desirable to match:

```
x = a + b + c
```

With an expression of the form:

```
x = cmlexpr(add1, +); // array of expressions called add1, joined by +
```

Retrieving a component could then be done with:

```
cmlexpr(add1[ 0 ]);:
```

Using generic items in the expr replicator would permit replacement of an operator. The example below would replace the original addition of `a`, `b` and `c` with a multiplication of the same:

```
a = expr(*; i = 0; i < cmlexpr(add1, length); i++; cmlexpr(add1[ i ]));
```
7.4.2 Flow Graph Matching

Limiting the language to AST matching requires transformations such as `fortowhile` to open up new opportunities for optimisation. Extending the language to allow matching on flow graphs would remove this need, as the conversion would have been carried out already.

7.4.3 Automatic vs User-Directed

The approach adopted here was one of automatic generation of a transformed solution. The Code Transformation Tool used an interactive approach, with the user deciding whether or not to apply an applicable transformation. A similar approach could be adopted with CML, providing a GUI to guide the user through the process of transforming their code.

The drawback with this alternative would be that the user must edit the transformed code, or go through the interactive process again. Storing meta-data in the source file to indicate which transformations were applied where may help to alleviate the problem, however only in areas where no changes have taken place.

7.4.4 Reversible Transformations

The transformations presented so far have been designed to reduce the number of cycles taken to execute and increase $f_{max}$ in the drive for faster execution. The transformation engine could equally be used for the reverse reason — transformations which remove parallelisation in an attempt to reduce power consumption. Indeed, many existing transformations could be run in reverse by swapping the pattern and generate sections of the transformation. The major difficulty with running transformations in reverse is ensuring data integrity.

Having transformations which attempt to optimise for all factors is possible in design exploration mode, as multiple options are generated and the optimal one for the situation can be chosen at the end. In single-solution mode, additional heuristics and user hints would be required to help determine which one of several options to apply.

7.4.5 Heuristics for Single Solution Mode

In single-solution mode, the transformation engine currently applies as many transformations as it can, in the order in which they are defined. A more useful approach would be to direct the transformation engine to optimise for speed, resources or power and have it apply heuristics to determine which transformations it should apply. Metadata attached to transformations could provide additional hints, such as whether a transformation is likely to increase power or resource usage, or decrease execution time. As shown in Chapter 6, it is difficult to predict this for some transformations.

Testing here indicates that parallelisation without hardware duplication increases power, has little effect on $f_{max}$, decreases cycle count and decreases resource usage slightly. More extensive testing may reveal further heuristics, however there is not enough data available at present.
7.5 Further Applications

Whilst the previous section discussed possible additions to the transformation engine directly, this section examines some possible applications beyond the realm of optimisation.

7.5.1 Cryptographic Systems

Looking beyond optimisations, the transformation system could play a key role in hardening against cryptographic attacks on FPGAs. Protection of encryption keys is vital to the security of a cryptographic system. Electronic systems such as FPGAs are susceptible to a variety of attacks[42].

The transformation system could be used to introduce artificial delays, duplication and obfuscation into the cryptographic algorithm in an attempt to modify the power signature sufficiently to protect against side-channel attacks.

7.5.2 The Aspect-Oriented Approach to Debugging

One major factor of digital system design debugging is being able to monitor signals inside the design to determine at which point it malfunctions. Technologies such as Altera’s SignalTap allow internal signals to be routed to spare pins on the chip during the FPGA place-and-route process.

One alternative possibility is to adapt to the transformation engine to permit definitions closer to the Aspect-Oriented model, introducing debugging signals at the source level.
7.6 Summary

This project has demonstrated that the pattern-matching approach to transformations can be especially effective with Handel-C, particularly with beginners who will tend to write inefficient C-style code. It can successfully restructure a program with a set of test transformations and improve performance by 70% in some cases.

It was shown that the effect of transformations varies between hardware targets, demonstrating that design-space exploration with real place-and-route tools is the most effective way to find out which transformations are best.

The limitations of the project as it stands have been discussed, with ideas for improvements and additional uses proposed.

Overall the transformation system demonstrates potential to be an extremely useful tool, particularly with the additional changes suggested.

The main contributions of this report are:

- A specification for a transformation description language for Handel-C (Chapter 3).
- A design and implementation of a parallelising transformation system for that language (Chapter 4).
- Automatic transformations capable of achieving 35-70% reduction in execution time, depending on circumstances (Chapter 5).
- An insight into the interaction of transformations, both with each other and with the platform their output runs on (Chapter 6).
Appendix A

Transformations

This appendix presents a variety of transformations created using the transformation engine.
A.1 Standard Transformations

The transformations in this section are always applied, but perform no optimisations in themselves. They serve only to restructure the code to enable further transformations.

A.1.1 std for single to seq

Convert a for loop with a single statement in the body to a for loop with a sequence of statements, containing one statement. This ensures that transformations matching for loops can do so in a consistent manner.

```cml
always transform std_for_single_to_seq {
    pattern {
        for (cmlstmt(init); cmlexpr(cond); cmlstmt(step))
        cmlstmt(body);
    }
    generate {
        for (cmlstmt(init); cmlexpr(cond); cmlstmt(step))
        {
            cmlstmt(body);
        }
    }
}
```

Validity Argument

From the language definition, both patterns are semantically identical.

A.1.2 std flatten par

The process of automatic parallelisation can generate nested parallel blocks. Leaving these in place can impede future matches or lead to unattractive output code.

```cml
always transform std_flatten_par {
    pattern {
        par {
            cmlstmtlist(preamble);
            par {
                cmlstmtlist(nested);
            }
            cmlstmtlist(postamble);
        }
    }
}
```
Validity Argument

From the language definition, both patterns are semantically identical, except in the case of local variables within the innermost block. The transformation engine does not allow transformations to destroy blocks with local variables.

A.1.3 std flatten seq

Transformations can generate nested sequential blocks. Leaving these in place can impede future matches or lead to unattractive output code.

```cml
always transform std_flatten_seq {  
    pattern {  
        {  
            cmlstmtlist(preamble);  
        }  
        cmlstmtlist(nested);  
        cmlstmtlist(postamble);  
    }  
}  
```

Validity Argument

From the language definition, both patterns are semantically identical, except in the case of local variables within the innermost block. The transformation engine does not allow transformations to destroy blocks with local variables.
A.2 Other Transformations

A.2.1 for to par

Automatically parallelise for loops with no loop-carried dependencies.

\[
\text{transform } \text{for}\_\text{to}\_\text{par} \{ \\
\text{cmlvar } i \; ; \; // \text{declare variables used} \\
\text{pattern} \\
\{ \\
\text{for}(i = \text{cmlexpr}(\text{init}); i < \text{cmlexpr}(\text{cond}); i += \text{cmlexpr}(\text{step})) \\
\{ \\
\text{cmlstmtlist}(\text{body}); \\
\} \\
\} \\
\text{generate} \\
\{ \\
\text{par}(i = \text{cmlexpr}(\text{init}); i < \text{cmlexpr}(\text{cond}); i += \text{cmlexpr}(\text{step})) \\
\{ \\
\text{cmlstmtlist}(\text{body}); \\
\} \\
\} \\
\text{conditions} \\
\{ \\
// \text{does not assign to the same location (this always fails!)} \\
\text{defs(}\text{cmlstmtlist}(\text{body})) \& \text{defs(}\text{cmlstmtlist}(\text{body})) = \{\}; \]

// no loop-carried dependencies
\text{uses(}\text{cmlstmtlist}(\text{body})) \& \text{defs(}\text{cmlstmtlist}(\text{body})) = \{\};

// no modifications to loop variable in body
\{i\} \& \text{defs(}\text{cmlstmtlist}(\text{body})) = \{\};

\text{constant(}\text{cmlexpr}(\text{cond})); // constant loop size
\text{constant(}\text{cmlexpr}(\text{step})); // constant step
\text{constant(}\text{cmlexpr}(\text{init})); // constant starting point

// the bounds are in the correct direction and produce at least 1 iteration
\text{eval}(((\text{cmlexpr}(\text{cond}) - \text{cmlexpr}(\text{init})) / \text{cmlexpr}(\text{step})) > 0);
\}
\}

A for loop with constant bounds and no loop-carried dependencies maybe parallelised.
A.2.2 factorise

An algebra manipulation example, illustrating factorisation to expose possibilities for constant folding and saving multipliers.

// x * a + x * b = x * (a + b)
transform factorise {
    pattern {
        cmlexpr(a) * cmlexpr(b) + cmlexpr(a) * cmlexpr(c)
    }
    generate {
        cmlexpr(a) * (cmlexpr(b) + cmlexpr(c))
    }
}

A for loop with constant bounds and no loop-carried dependencies maybe parallelised.

A.2.3 deadcode

Dead code elimination example.

transform {
    cmlvar i;
    pattern {
        cmlstmtlist(pre);
        i = cmlexpr(dead);
        cmlstmtlist(mid);
        i = cmlexpr(b);
        cmlstmtlist(post);
    }
    generate {
        cmlstmtlist(pre);
        cmlstmtlist(mid);
        i = cmlexpr(b);
        cmlstmtlist(post);
    }
    conditions {
        // i is not used in mid (before it is redefined)
        uses(cmlstmtlist(mid)) & defs(i = cmlexpr(b);) == {};
    }
}
A.2.4 dependencies

Additional parallelisation is possible by removing dependencies. In this example a dependency is removed by duplicating the expression assigned to a variable. This opens up potential for the \( i \) and \( j \) statements to be parallelised.

\[
\text{transform } \{ \\
\text{cmlvar } i, j; \\
\text{pattern} \\
\{ \\
\text{cmlstmtlist}(\text{pre}); \\
j = \text{cmlexpr}(a); \\
i = \text{cmlexpr}(b) + j; \\
\text{cmlstmtlist}(\text{post}); \\
\} \\
\text{generate} \\
\{ \\
\text{cmlstmtlist}(\text{pre}); \\
j = \text{cmlexpr}(a); \\
i = \text{cmlexpr}(b) + \text{cmlexpr}(a); \\
\text{cmlstmtlist}(\text{post}); \\
\}
\]

Appendix B

Transformation Language Syntax

This appendix contains a subset of the CML syntax in EBNF form, modified directly from the ANTLR grammar. The full statement syntax is not included.

statement and expression productions in this syntax refer to valid Cobble/Handel-C statements and expressions and the CML productions cml_gen_stmt, cml_gen_stmtlist and cml_gen_expr.
ID = ('A'−'Z' | 'a'−'z')+ ('A'−'Z' | 'a'−'z' | '_')*
cml ::= ( "always" ) "transform" {ID} "(" custom_transform ")"
  )*
custom_transform ::= {
cml_declarator
  "pattern" cml_stmt_list
  "generate" cml_stmt_list
  { "conditions" ":" cml_conditions "}" } } 
cml_stmt_list ::= 
  { " ( statement | cml_gen_stmtlist ";" )* | expression } 
"}"
cml_gen_stmt ::= "cmlstmt" "(" ID ")"
cml_gen_stmtlist ::= "cmlstmtlist" "(" ID ")"
cml_expr ::= "cmlexpr" "(" ID ")"
cml_declarator ::= "cmlvar" { cml_declarator_list } ";" 
cml_declarator_list ::= cml_var_decl ( ":" cml_var_decl)*
cml_var_decl ::= ID ( ":" ["]" )*
cml_conditions ::= ( "!" cml_condition ";:" | cml_condition ";:" )*
cml_condition ::= ( cml_setconstraint | cml_exprconstraint )
cml_exprconstraint ::= ID "(" expression ")"
cml_primaryset ::= ( cml_explicitset | cml_usedset | cml_defset )
cml_setconstraint ::= cml_setintersection "==" cml_setintersection

cml_setunion ::= cml_setintersection ( ":&" cml_setunion )*
cml_explicitset ::= "{" {ID ( ":" ID)* } "}"
cml_usedset ::= "uses" "(" ( statement | cml_gen_stmtlist ) ")"
cml_defset ::= "defs" "(" ( statement | cml_gen_stmtlist ) ")"
cml_gen_stmts ::= ( cml_gen_stmt | cml_gen_stmtlist )
Appendix C

Source Listings

C.1 Matrix Multiply Source

This section contains the unoptimised and fully optimised versions of the Matrix Multiply example used in Chapter 6. The original version is the Cobble hand-coded design, while the fully optimised version has been generated by the transformation engine.
C.1.1 Original Matrix Multiply

```c
unsigned int 32 Out1;
unsigned int 1 Finished = 0;

interface pebble ports(Out1, Finished) (unsigned int 32 In1,
                   unsigned int 32 In2);

unsigned int 32 aa[3][3];
unsigned int 32 bb[3][3];
unsigned int 32 cc[3][3];

unsigned int 32 loadBufA[9];
unsigned int 32 loadBufB[9];

unsigned int 2 i, j, k;
unsigned int 4 loadi = 0;

void main()
{
    /* Strictly timed input section – matrix elements are read 1 per cycle */
    while(loadi != 9)
    {
        par
        {
            loadi++;
            loadBufA[loadi] = In1;
            loadBufB[loadi] = In2;
        }
    }
    loadi = 0;
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            aa[i][j] = loadBufA[loadi];
            bb[i][j] = loadBufB[loadi];
            cc[i][j] = 0;
            loadi++;
        }
    }
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            for(k = 0; k < 3; k++)
            {
```
\[
cc[i][j] = cc[i][j] + aa[i][k] \times bb[k][j];
\]

loadi = 0;
for (i = 0; i < 3; i++)
{
  for (j = 0; j < 3; j++)
  {
    loadBufA[loadi] = cc[i][j];
    loadi++;
  }
}

// Strictly timed output section - matrix elements are emitted 1 per cycle
loadi = 0;
while (loadi != 9)
{
  par
  {
    loadi++;
    Out1 = loadBufA[loadi];
  }
}

Finished = 1;
C.1.2 Optimised Matrix Multiply

```c
void main()
{
    while ((loadi != 9))
    {
        par
        {
            loadi = (loadi + 1);
            loadBufA[(9 - 1)] = In1;
            loadBufB[(9 - 1)] = In2;
        }
        par (q = 0; (q < (9 - 1)); q = (q + 1))
        {
            loadBufA[(q <= 4)] = loadBufA[((q + 1) <= 4)];
            loadBufB[(q <= 4)] = loadBufB[((q + 1) <= 4)];
        }
    }
    loadi = 0;
    {
        i = 0;
        while ((i != 3))
        {
            j = 0;
            while ((j != 3))
            {
                par
                {
                    aa[i][j] = loadBufA[loadi];
                    bb[i][j] = loadBufB[loadi];
                    cc[i][j] = 0;
                    loadi = (loadi + 1);
                    j = (j + 1);
                }
            }
        }
        i = (i + 1);
    }
}
{
    unsigned int 32 t0;
    unsigned int 32 t1;
    unsigned int 32 t2;
    {
        i = 0;
        while ((i != 3))
        {
            j = 0;
```
while ((j != 3))
{
    par
    {
        t0 = (aa[i][0] * bb[0][j]);
        t1 = (aa[i][1] * bb[1][j]);
        t2 = (aa[i][2] * bb[2][j]);
    }
    par
    {
        cc[i][j] = ((t0 + t1) + t2);
        j = (j + 1);
    }
    i = (i + 1);
}
loadi = 0;
{
    i = 0;
    while ((i != 3))
    {
        j = 0;
        while ((j != 3))
        {
            par
            {
                loadBufA[loadi] = cc[i][j];
                loadi = (loadi + 1);
                j = (j + 1);
            }
            i = (i + 1);
        }
        loadi = 0;
    }
while ((loadi != 9))
{
    par
    {
        loadi = (loadi + 1);
        Out1 = loadBufA[loadi];
    }
    Finished = 1;
}
Glossary

abstract syntax tree (AST)
A tree representation of the source of a program.

application-specific integrated circuit (ASIC)
An integrated circuit containing an application-specific design.

behavioural synthesis
The mechanism where untimed high-level code is converted into a timed circuit design
by the compiler, based on constraints given by the user.

Cobble Meta Language (CML)
A transformation description language based on CTT, described but never implemented
by Tim Todman. The inspiration for the transformation system described in this report.

data flow graph (DFG)
A graph representing the data dependencies between operations in a program.

$f_{max}$
The maximum clock frequency a design can run at.

field-programmable gate array (FPGA)
A logic device containing reprogrammable logic cells with reprogrammable interconnections.

Handel-C
A proprietary C-based language for the high level description of digital system designs.

“inefficient hardware”
Used within this report to describe hardware where there exists an alternative with
better speed, resource or power characteristics without affecting the other factors.

intermediate representation (IR)
A generic term used to describe the format of a program in the intermediate stages of
processing in a compiler. This is often a tree, however may also be a linear representa-
tion such as single static assignment.
Logic Element

The unit used by Altera to measure resource usage on their FPGAs.

place and route

The process of taking a digital system design and assigning it to components on an FPGA, then routing signals between the components.

Slice

The unit used by Xilinx to measure resource usage on their FPGAs.

structural synthesis

The mechanism where strictly timed high-level code is converted into a circuit design by the compiler, respecting the cycle discipline the language defines.

Transformation subset of the Cobble Meta Language (T-CML)

The variation of Todman's CML described in this report.

Very large scale integration Hardware Description Language (VHDL)

A language for the description and modelling of digital systems.
Bibliography


