

# **Dataflow Analysis and Optimization of High Level Language Code for Hardware-Software Co-design**

by

**R. Brendan O'Connor**

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE**

in

**Electrical Engineering**

APPROVED:

Peter M. Athanas, Chairman

A. Lynn Abbott

Nathaniel J. Davis, IV

May 11, 1996

Blacksburg, Virginia

Keywords: CCM, Co-Design, Dataflow, Hardware Compilation

Copyright 1996, R. Brendan O'Connor

# **Dataflow Analysis and Optimization of High Level Language Code for Hardware-Software Co-design**

by

R. Brendan O'Connor

Committee Chairman: Peter M. Athanas

Electrical Engineering

## **(ABSTRACT)**

Recent advancements in FPGA technology have provided devices which are not only suited for digital logic prototyping, but also are capable of implementing complex computations. The use of these devices in multi-FPGA Custom Computing Machines (CCMs) has provided the potential to execute large sections of programs entirely in custom hardware which can provide a substantial speedup over execution in a general-purpose sequential processor. Unfortunately, the development tools currently available for CCMs do not allow users to easily configure multi-FPGA platforms. In order to exploit the capabilities of such an architecture, a procedure has been developed to perform a dataflow analysis of programs written in C which is capable of several hardware-specific optimizations. This, together with other software tools developed for this purpose, allows CCMs and their host processors to be targeted from the same high-level specification.

## **ACKNOWLEDGEMENTS**

I would like to thank Dr. Peter Athanas for his constant encouragement throughout the duration of this project. I would also like to thank Dr. Abbott and Dr. Davis for their support and for their time.

I would like to thank Jim Peterson for his help throughout this project. I would also like to thank my family for their consistent support throughout my academic career. In addition I would like to thank all of my friends for their continued support. Finally, a special thanks to my sister, Amalie, who has consistently encouraged me to do my best.

# TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Custom Computing Machines . . . . .	1
1.2	Current Computing Solutions . . . . .	3
1.2.1	VLSI Technology . . . . .	3
1.2.2	System Architectures . . . . .	4
1.3	Advantages of Custom Computing Machines . . . . .	5
1.3.1	The Application Development Problem . . . . .	5
1.3.2	The Solution . . . . .	7
1.4	Research Goal . . . . .	8
1.4.1	Research Contributions . . . . .	8
1.5	Thesis Organization . . . . .	9
<b>2</b>	<b>Background</b>	<b>10</b>
2.1	Overview of Previous Work . . . . .	11
2.1.1	The C Hardware Description Languages . . . . .	11
2.2	HDL Based Approach to Hardware Synthesis . . . . .	14
2.2.1	Timing Information . . . . .	14

## CONTENTS

2.3	HLL Based Approach to Hardware Synthesis: The <i>PRISM</i> Model . . . . .	16
2.3.1	C Language Input . . . . .	18
2.3.2	Architecture Descriptions and Libraries . . . . .	18
2.4	Overview of this method . . . . .	19
<b>3</b>	<b>Approach</b>	<b>21</b>
3.1	Compilation . . . . .	22
3.2	Directed Acyclic Graphs . . . . .	23
3.2.1	Representation of Control Flow Information . . . . .	24
3.3	Symbolic Operations . . . . .	25
3.3.1	Type Information . . . . .	27
3.4	Program Representation . . . . .	29
3.5	Graph Description Language . . . . .	29
<b>4</b>	<b>GDLGEN</b>	<b>32</b>
4.1	Program Representation . . . . .	33
4.2	Parsing . . . . .	34
4.2.1	Global Symbols and Variables . . . . .	34
4.2.2	Functions . . . . .	35
4.2.3	Code . . . . .	37
4.3	Global Symbol Resolution . . . . .	37
4.4	Hardware and Software Pool Allocation . . . . .	38

## CONTENTS

4.5	Label Merging . . . . .	40
4.6	Register and Memory Allocation . . . . .	40
4.6.1	Registered Variables . . . . .	40
4.6.2	Memory Variables . . . . .	41
4.7	Block Merging . . . . .	43
4.8	Analysis of Control-Flow Information . . . . .	43
4.8.1	Conditional Branches . . . . .	43
4.9	Unconditional Branches . . . . .	47
4.10	The Switch-Case Construct . . . . .	47
4.11	GDL Emission . . . . .	49
4.12	Modifications to <i>lcc</i> . . . . .	49
<b>5</b>	<b>Optimization Strategy</b>	<b>50</b>
5.1	Multiplication Strength Reduction . . . . .	51
5.1.1	Multiplication Chains . . . . .	51
5.1.2	Replacement . . . . .	53
5.2	Shift Strength Reduction . . . . .	54
5.3	Balancing of Arithmetic DAGs . . . . .	55
<b>6</b>	<b>Experimental Results</b>	<b>57</b>
6.1	A Simple Example . . . . .	57
6.2	A More Complex Example . . . . .	58

*CONTENTS*

<b>7</b>	<b><i>GDLGEN</i> Tool Organization</b>	<b>64</b>
7.1	General Structure . . . . .	64
7.2	Program Flow . . . . .	65
<b>8</b>	<b>Conclusions</b>	<b>69</b>
8.1	Successes of the tool . . . . .	69
8.2	Shortcomings of the tool . . . . .	70
8.3	Future work . . . . .	70
	<b>References</b>	<b>71</b>
	<b>Vita</b>	<b>73</b>

## LIST OF FIGURES

1.1	The Specification Pyramid. . . . .	6
1.2	CCM application development times. . . . .	7
2.1	Common subsets of C language implemented by hardware compilers. . . . .	11
2.2	Example VHDL dual-ported register model. . . . .	15
2.3	Compilation of HLL code to a custom computing machine. . . . .	16
3.1	Trivial C program. . . . .	23
3.2	DAG generated for <code>(*j)++</code> ; . . . . .	23
3.3	Linearized DAG for <code>(*j)++</code> ; . . . . .	24
3.4	Valid type conversions. . . . .	27
3.5	Conversion sample C program. . . . .	28
3.6	Integer to Unsigned DAG. . . . .	28
3.7	Forest for trivial C program. . . . .	29
3.8	Sample GDL Code. . . . .	30
4.1	<i>Gdngen</i> program structural representation. . . . .	33
4.2	A simple C program. . . . .	34
4.3	Structure of the <code>Program</code> object. . . . .	35

LIST OF FIGURES

4.4	Symbolic output file. . . . .	36
4.5	A C program which will demonstrate the hardware/software allocation. . . . .	38
4.6	GDL translation showing generated function calls. . . . .	39
4.7	Example C program containing pointer manipulations. . . . .	41
4.8	GDL code showing pointer manipulations. . . . .	42
4.9	Program block structure for <i>if-else</i> construct. . . . .	44
4.10	Program block structure for <i>while</i> loop. . . . .	46
4.11	Program block structure for a switch-case statement. . . . .	48
5.1	Multiplication chain example C program. . . . .	51
5.2	Multiplication chain GDL code example . . . . .	52
5.3	Histogram of chain lengths up to 10. . . . .	53
5.4	An unbalanced dataflow graph. . . . .	55
5.5	An equivalent balanced dataflow graph. . . . .	56
6.1	Simple C code example. . . . .	58
6.2	Simple GDL code example. . . . .	59
6.3	A simple schedule. . . . .	60
6.4	A C program to calculate a Mandelbrot set. . . . .	61
6.5	A GDL translation of the Mandelbrot program. . . . .	62
6.6	A schedule generated for the Mandelbrot GDL code. . . . .	63
7.1	The structure of the <i>gdlgen</i> tool. . . . .	67

## LIST OF TABLES

3.1	LCC control flow symbolic operations. . . . .	24
3.2	LCC symbolic type suffixes. . . . .	25
3.3	LCC datapath symbolic operations. . . . .	26
3.4	LCC variable access symbolic operations. . . . .	27

# Chapter 1

## Introduction

General-purpose microprocessors have been unable to provide the performance required by the most challenging problems in the areas of real-time image processing, cryptography, and simulation acceleration, among other areas [6]. As a result, these problems have long been in the realm of supercomputers; however, the costs associated with supercomputers can be prohibitive. A number of *Custom Computing Machines* (CCMs), architectures which provide reconfigurable processing and interconnect capability, have been designed to provide computational power which rivals the supercomputers but at a fraction of the cost. A major hurdle to the widespread use of CCMs as a computing solution is the difficulty inherent in the application development process. The goal of this research is to provide a comprehensive tool for CCM design automation using a *standard* application development environment, the ANSI-C language. This tool, the *PRISM* compiler, can be used to bridge the gap between traditional software development and CCM application development.

### 1.1 Custom Computing Machines

There are several key features that can generally be found in a Custom Computing Machine (CCM).

## CHAPTER 1. INTRODUCTION

1. Reconfigurable processing elements (PEs). In order to allow users to specify custom architectures, CCMs must contain some sort of reconfigurable logic resource that can be used to implement the architecture. In most modern CCMs, FPGAs fill this role.
2. Memory. A memory system for storing data.
3. Reconfigurable communications resources. In Multi-PE architectures, the need to communicate between PEs is crucial. CCMs address this communications requirement by having reconfigurable communications resources.
4. I/O. CCMs also need a mechanism to both communicate with their host processors, and to be programmed.

These elements provide a platform which can be configured to implement custom hardware architectures to execute user applications. While general-purpose microprocessors have a fixed architecture, CCMs allow the developer to *specify* an architecture uniquely suited to solving a particular problem.

There are many examples of CCMs developed for both research and commercial purposes. The Splash-2 system which was developed by the Supercomputing Research Center is a fairly common CCM platform consisting of 16 FPGAs each coupled with 512KB (256Kx16) of Static RAM and a configurable crossbar for communications [3]. Annapolis Micro Systems [29] makes a commercial version of the Splash-2 architecture called the Wildfire. Systems are also available from Giga Operations [30], and Virtual Computer Corporation [31].

## CHAPTER 1. INTRODUCTION

### 1.2 Current Computing Solutions

Moore's Law dictates that the performance of microprocessors doubles approximately every eighteen months. This has held true over the past twenty years. However, as the cost of building the advanced fabrication plants that are required to keep this pace increases, many companies will find it difficult to justify the expense [20]. Even at this dramatic pace, general purpose microprocessors have not been able to provide the performance required for the most computationally intensive applications. The performance increases in commercial microprocessors is a result of the advancement of VLSI technology which allows processor architectures which include more hardware support for specialized user applications. In this, CCMs are still ahead of microprocessors because hardware can be custom-tailored to accelerate user applications.

#### 1.2.1 VLSI Technology

Over the past two years, the "flagship" technology being used in commercial processors was on the order of 0.6-0.8  $\mu\text{m}$ . Today, many commercial processors are being implemented in 0.25-0.35  $\mu\text{m}$  processes with 0.1  $\mu\text{m}$  processes on the way. This results in not only a savings in area but a savings in die cost for processor manufacturers. This trend also provides denser chips. More transistors per unit area enables more and more radical architectures to be explored while still maintaining commercial production feasibility.

## CHAPTER 1. INTRODUCTION

### 1.2.2 System Architectures

As a result of the increased transistor budgets, processor designers have turned their attention towards more advanced architectures and larger on-chip caches. Currently, designers are pushing super-scalar architectures to their practical limits. Processors capable of retiring four instructions per clock cycle are becoming fairly common [26, 27, 28]. Designers are also exploring and implementing designs that allow the processor the ability to reorder the program instruction stream through careful in-hardware analysis of control and data dependencies. In addition, designers are now beginning to incorporate special-purpose hardware units to speed application execution. A good example of this trend is Sun's UltraSPARC. The UltraSPARC contains a family of instructions designed to handle common multimedia and graphics operations efficiently [28].

There is a limit to what can be done in a general-purpose architecture to provide hardware support for specific operations, however. Microprocessor designers must keep their designs cost-effective, and must therefore make careful decisions on which operations to provide hardware support. Microprocessors must also be able to handle a broad range of applications. Since generality and efficiency are inversely related to one another, the performance of general-purpose microprocessors will be acceptable on most applications, but rarely exceptional [6]. CCMs have found their niche in the high-performance computing area because they offer the flexibility to develop architectures which can provide exceptional performance on the particular problem they were designed for.

### 1.3 Advantages of Custom Computing Machines

CCMs, have a high potential to speedup user applications. The primary reason for the potential speedup is the ability of the user to develop custom hardware architectures. Instead of executing algorithms iteratively as instructions on a general purpose microprocessor, algorithms can be executed in parallel hardware on the CCM. Given the finite resources available on a CCM platform, large applications may not be able to fit in the hardware. This limitation introduces the need for run time reconfiguration (RTR).

RTR allows portions of the hardware to be reconfigured while other portions continue computation. RTR introduces the concept of *virtual hardware* which is similar to virtual memory for conventional computers. Virtual memory allows applications larger than the size of main memory to be executed by swapping blocks of memory to secondary storage. Virtual hardware allows applications which consume more hardware resources than are available to be executed by swapping blocks of hardware in and out of portions of the CCM platform. In light of all of these advantages, the design methodologies for CCM platforms, leave much to be desired. The *PRISM* compiler attempts to address the issue of CCM application design by providing a common interface across hardware and software.

#### 1.3.1 The Application Development Problem

The development of custom architectures to solve computing problems requires knowledge about both hardware design and the specifics of the particular CCM platform being used. Currently, most CCM applications are developed using a structural design methodology. This can limit the reuse

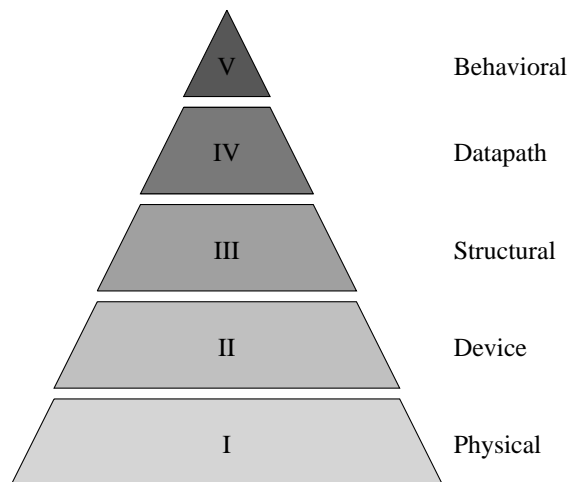


Figure 1.1: The Specification Pyramid.

of application code because of differing temporal and spatial requirements between applications. In addition, since the structural design methodology requires detailed architectural knowledge, major problems can be encountered when porting an application to a new platform. Although HDLs provide the facility to use higher-level *behavioral* specifications for application development, it can be difficult to make accurate resource estimates from these specifications. Figure 1.1 shows the various levels at which applications can be developed. The higher up the pyramid, the more abstract the design, and hence the greater the reliance on the development tools. Level V designs are purely behavioral and contain no temporal or spatial information. Level IV designs are target independent, but contain explicit data-path operators. Level III designs are structural designs in which both the temporal and spatial properties of the design are explicitly specified. Level II designs are specified at the device level, while Level I designs are at the physical level.

The difference in development times for a software application and an equivalent hardware implementation can be significant [18]. This is largely due to the abstraction that is provided when

## CHAPTER 1. INTRODUCTION

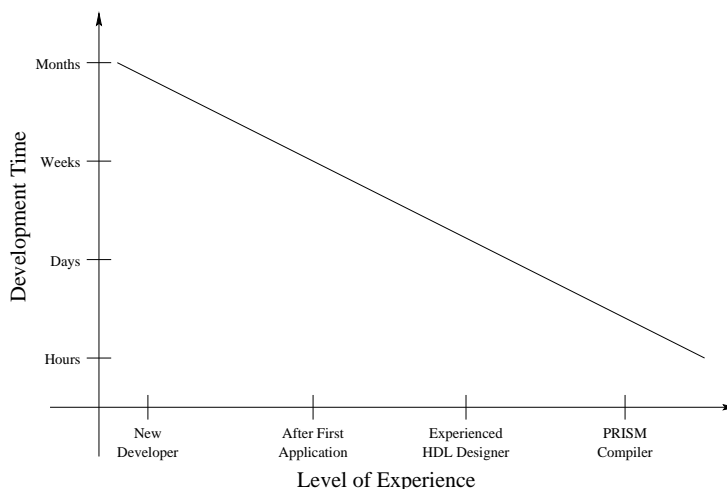


Figure 1.2: CCM application development times.

working in a high level language like C. In order to develop useful CCM applications, developers must also be hardware designers. Even with knowledge of hardware design, developers still face a significant challenge in developing CCM applications using modern Hardware Description Languages (HDLs) in a timely fashion due to the architectural issues that must be considered. In fact, development time for CCMs using current techniques can be on the order of several times longer than for a pure software application to perform the same function. Figure 1.2 shows the relationship between both the proficiency of the designer and the tools used and application development time.

### 1.3.2 The Solution

Enabling CCMs to be targeted by application developers as easily as a general-purpose microprocessor is essential if they are to become a serious computing solution. In order to achieve this, there must be a *standard* interface for both CCMs and microprocessors. The ANSI-C language has become the standard for software applications and is commonly used in the development of

## CHAPTER 1. INTRODUCTION

portable applications. For this reason, ANSI-C was chosen as the input language to the *PRISM* compiler. Developers can now use a single code base to target both pure software, CCM, and hybrid software/CCM applications. Unfortunately, compilation of HLL programs becomes a completely different problem since the underlying machine model has been removed [6]. The *PRISM* compiler addresses the CCM development problem by providing a familiar interface to the programming of CCMs making them accessible to as many developers as possible.

### 1.4 Research Goal

In order to accomplish the task of compiling ANSI-C programs into hardware and software for CCMs and their host processors, C programs must first be converted to a graph description language (GDL). The goal is to be able to translate any ANSI-C program into an equivalent GDL representation. The GDL is subsequently interpreted by the scheduler and partitioner (*SAS*) in order to generate a hardware image for a CCM. This thesis describes a tool to generate this graphical description. This tool, along with the *lcc* freeware C compiler [8] generates a GDL description of an input program, performs hardware specific optimizations, and does preliminary hardware-software partitioning.

#### 1.4.1 Research Contributions

My work has contributed to the project in the following ways:

- Provided a dataflow analysis tool to generate a GDL description of ANSI-C programs.
- Generated a library of functional units for use with the compiler.

## CHAPTER 1. INTRODUCTION

### 1.5 Thesis Organization

This thesis describes a tool, *gdlgen*, which analyzes and optimizes a graphical description of an ANSI-C program for implementation on a CCM platform. The tool reads a graphical description of a C program and performs both hardware specific optimizations as well as preliminary hardware-software partitioning. The output from this tool is fed to the *SAS* (Simulated Annealing Scheduler) tool for scheduling and partitioning.

Chapter 2 describes previous work done in this area. In addition, the *gdlgen* tool is presented and its objective is outlined. Chapter 3 describes the approach taken and the tools used to develop the *gdlgen* tool. The operation of *lcc* is discussed to introduce the fundamental data structures used by the tool. The use of the symbolic dump from *lcc*, which is used as the input to the tool is described. In Chapter 4, the implementation of the tool is discussed. Segmentation of programs into hardware and software pools is presented, along with the allocation of registers. The analysis of control flow information is presented, along with the matching of control flow information to GDL constructs. Chapter 5 describes the motivation and implementation of specific optimizations. Specifically, the reduction of constant operand shifts and multiplications is examined from a hardware implementation perspective. The implications of these optimizations on the scheduler is also discussed. Chapter 6 contains experimental results obtained through the use of the *gdlgen* and *SAS* tools. Graphical schedules are presented for several example C programs. Chapter 7 gives an overview of the structure of the *gdlgen* tool. An algorithmic flow of the tool is also presented at a high level. Chapter 8 concludes this thesis, and discusses the weaknesses in the tool as an incentive for future work in this area.

# Chapter 2

## Background

Automatic generation of hardware and software descriptions of CCM applications is a non-trivial task. Until now, the primary means of specifying the configuration of CCMs was through the use of Hardware Description Languages (HDLs). HDLs are used to manually describe the hardware and the partitioning of the design onto the target architecture. In addition, control and communications software has to be written to interact between the host computer and the CCM platform.

There have been several attempts at making hardware compilers for CCMs that have begun to use high level languages (HLLs) as their inputs. Unfortunately, many of these attempts have actually transformed the input language from a tool for HLL development into another HDL [11, 13, 10]. These attempts have provided hardware designers with familiar syntax in which to represent their designs, but have fallen short of allowing designers flexibility provided by HLLs by providing support for a subset of the input language. In addition, previous compilers have not addressed the codesign issue. In these respects, the *PRISM* compiler is truly novel. Any ANSI-C program is valid as input to the compiler, and both hardware and software descriptions are generated.

This chapter outlines some of the previous attempts at synthesizing hardware from HLL descriptions. Several previous hardware compilers are introduced, and their features are examined.

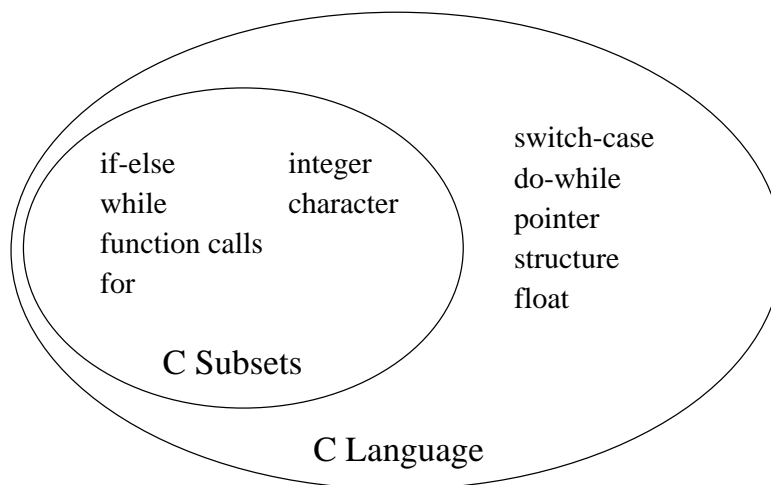


Figure 2.1: Common subsets of C language implemented by hardware compilers.

Additionally, current CCM design methodologies are introduced as is the *PRISM* design flow.

## 2.1 Overview of Previous Work

Much of the previous work in this area has involved new synthesis techniques for subsets of common HLLs. Different systems handle different subsets of the languages. In order to simplify compilation, these compilers reduce the number of control flow constructs that are available to the developer and may limit the number of data types that a user can use.

### 2.1.1 The C Hardware Description Languages

Because of the incredible flexibility of the C language, programmers have taken to it for a number of tasks. Because of this same flexibility however, problems arise when trying to synthesize hardware descriptions from C programs. The number of control flow constructs complicates the analysis of HLL code significantly, as does the ability to access variables through pointers. In

## CHAPTER 2. BACKGROUND

addition, the many types and sizes of variables can cause difficulty especially when structures and unions are used. These difficulties have been overcome by implementing subsets of the C language, and restricting valid data types. Figure 2.1 shows the commonly implemented subsets of the C language.

Many parameters necessary for generating hardware specifications are also missing in C program specifications. Most notably, partitioning and temporal information which is explicitly specified in HDLs is absent in C. As a result, either this information must be specified explicitly, or it must be determined by the compiler. Many previous C hardware compilers overcome this obstacle by requiring explicit partitioning information as in HDLs [13, 10], or by assuming a static target architecture [24].

These restrictions are not without consequence, however. Because of the partitioning restrictions, applications cannot be ported easily to new CCM architectures because either the code is target-specific or the compiler is target-specific. In addition, the implementation of language subsets causes difficulty in portability between compilers. In order to make C a valid language in which to develop CCM applications, these issues must be addressed.

### Control Flow Restrictions

Most of these compilers including [11, 13, 10] only handle a subset of the control flow constructs. These compilers typically handle `if` and `while` constructs, and prohibit the use of any other control flow constructs. The authors of [11] do not allow the use of *any* explicit control flow constructs. Clearly, the more restrictive a language is, the less likely it will be used.

## CHAPTER 2. BACKGROUND

### Data Type Restrictions

Many of these compilers also limit the number of data types that the developer has access to. In fact, [13] does not allow any datatype other than its `bitvector` type. Compilers [24, 11, 10] do not allow more advanced data types such as pointers and structures. Again, these compilers have placed significant restrictions on the input language.

### Architectural Knowledge Restrictions

Another restriction that has been commonly made by previous compiler efforts is the requirement of architectural knowledge as either a part of the input program or built-in to the compiler. Compilers that require partitioning information as part of user programs make it more difficult for developers to target hardware platforms [13]. This causes the user to partition designs manually across the CCM platform as in typical HDL-style application development. At the same time, compilers which have built-in knowledge of a particular platform can easily become outdated, and lack the flexibility to provide a general-purpose CCM application development environment [24].

### Codesign Restrictions

Another restriction present in previous compilers is similar to current CCM design practices. Developers are required to specify the software which runs on the host processor separately from the hardware which is implemented on the CCM. The need to develop both hardware and software separately significantly increases the complexity of developing a CCM application.

## CHAPTER 2. BACKGROUND

### 2.2 HDL Based Approach to Hardware Synthesis

Currently, CCM application designers use a number of standard HDLs to implement their designs. An advantage to this approach is the ability to work at varying levels of abstraction. The designer can specify in detail specialized combinational circuitry as well as specifying higher-level constructs where speed and or area are not critical.

Partitioning of a design over the processing elements typically found in current-generation CCMs requires a considerable amount of experience with different technologies to accurately estimate the amount of logic that can be placed on a chip with limited resources [25]. The resource issue is generally a two-part problem involving both the logic and routing resources of the PE. Estimating the amount of logic necessary to implement a certain HDL design is not a trivial task. However, an accurate estimate of logic resources can be determined if the design is implemented structurally. Estimating logic resources for behavioral designs can be extremely difficult because of the dependence on the synthesis tool. The amount of routing resources required by a design can be much more difficult to estimate. The routing resources required by a design can vary significantly based on how the design is expressed in the HDL.

#### 2.2.1 Timing Information

Hardware description languages make explicit many pieces of information which are not available in typical HLLs. A prime example of this is the timing information that is an integral part of any HDL model of a system. HDLs do this because in hardware systems, signal timing is absolutely critical. In fact, HDLs always keep track of explicit timing information for all signals in a piece of

## CHAPTER 2. BACKGROUND

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

--Entity declaration. Define the interface to this module.
ENTITY reg IS
PORT(
    clk: in std_logic;
    reset:in std_logic;
    asel: in std_logic;
    bsel: in std_logic;
    csel: in std_logic;
    aout: out std_logic_vector(15 downto 0);
    bout: out std_logic_vector(15 downto 0);
    cin: in std_logic_vector(15 downto 0)
);
END reg;

--Architecture description of the module.
ARCHITECTURE newreg OF reg IS
SIGNAL this_reg: std_logic_vector(15 downto 0);
BEGIN

--Execute this block whenever an event occurs on clk or reset.
PROCESS(clk,reset)
BEGIN
    if (reset = '1') then
        this_reg <= "0000000000000000";
    end if;
    if (clk'event AND clk = '1') then
        if(csel = '1') then
            this_reg <= cin;
        end if;
    end if;
END PROCESS;

--The following code is always executed concurrently.
aout <= this_reg WHEN asel = '1' ELSE "ZZZZZZZZZZZZZZZZ";
bout <= this_reg WHEN bsel = '1' ELSE "ZZZZZZZZZZZZZZZZ";
END;
```

Figure 2.2: Example VHDL dual-ported register model.

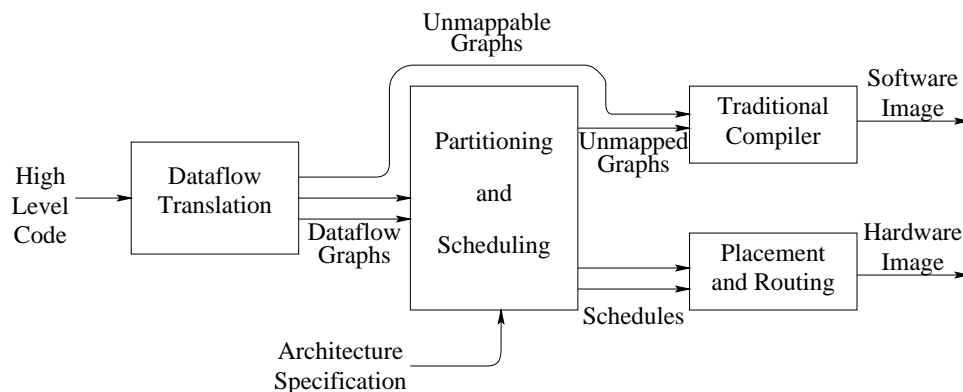


Figure 2.3: Compilation of HLL code to a custom computing machine.

code [2]. In addition, HDLs allow considerable flexibility in the way that input code can actually be implemented. Sections of code in HDLs are frequently executed concurrently without being explicitly specified as such by the designer. This is one of the reasons that HDLs take such care to keep timing information. In HLLs, since all statements execute sequentially, timing information is irrelevant. In VHDL, statements which are part of an `architecture` are executed concurrently, while statements which are part of a `process` or a subprogram are executed sequentially [2]. Because this is a different “mindset” than HLL systems where code is executed in order, HDLs can be more difficult to work with. Since designers also need to design systems which are not only functionally correct, but which satisfy timing constraints, the complexity of the HDL code increases. Figure 2.2 shows a complete VHDL model of a dual-ported register. Note that all statements within the `PROCESS` block are executed sequentially, while all statements not in the `PROCESS` block are executed concurrently. In this case, the assignments to `aout` and `bout` are done concurrently.

### 2.3 HLL Based Approach to Hardware Synthesis: The *PRISM* Model

## CHAPTER 2. BACKGROUND

As CCMs become more advanced, there is a trend to view them not only as special-purpose configurable computers, but also as high-speed reconfigurable coprocessors. As such, CCMs need to have a development environment that is familiar to developers. For this reason, ANSI-C was chosen as the input to the *PRISM* compiler.

Two items must be specified as input to the compiler: the high-level language source code, and a specification of the architecture of the CCM. Architecture descriptions are totally reusable, and need only be generated once for a particular CCM platform. The compiler begins by transforming the high-level code into a collection of dataflow graphs [8]. These dataflow graphs are then analyzed, optimized and translated into a Graph Description Language (GDL) format which is used by the rest of the compiler. The compiler must then schedule and partition the graphs so that they fit into the available hardware of the CCM. This is accomplished by viewing each PE as a *resource pool*. Functional units can be allocated in a resource pool as long as that pool is not fully utilized. After scheduling and partitioning have completed, schedules are passed to placement and routing tools to generate hardware images for the PEs of the CCM. Graphs which can not be made to fit, due to either the inclusion of an operation not supported by the CCM (e.g. file operations) or the limited amount of resources available, must be compiled into software to be run on the host computer [19]. In addition, operations (i.e. IEEE floating point operations) which are too resource intensive to be implemented in hardware, or for which there is no functional unit can be designated to be implemented in software only. These graphs are passed to a traditional compiler to be compiled into an executable software image. Figure 2.3 shows the general flow from the original source code written in a high-level language to the resulting hardware and software images used to program

## CHAPTER 2. BACKGROUND

the CCM and its host computer.

### 2.3.1 C Language Input

Since the C language has become a standard development tool for all types of general-purpose computing platforms and operating systems, it is a natural input language for a new development tool. Since the compiler supports the complete ANSI-C language, porting to a CCM platform is a relatively simple matter. Because of the simplicity of porting, *existing* as well as new applications can be targeted to CCM platforms. This provides performance improvements to a large number of programs with a limited hardware investment.

### 2.3.2 Architecture Descriptions and Libraries

To the developer, the *PRISM* compiler looks exactly like a standard ANSI-C compiler with few exceptions. In order to target a specific CCM, the compiler simply needs an architectural description of the hardware platform which outlines the resource and communications capabilities of the platform. Additionally, the linker requires a library of functional units that can be instantiated on the hardware platform as well as a description of their time and size requirements. Both of these additional requirements need only be produced once, unless the platform or library capabilities change.

## CHAPTER 2. BACKGROUND

### 2.4 Overview of this method

This thesis describes a tool which performs hardware optimizations, control flow analysis, and preliminary hardware-software allocation in order to generate a graph description language (GDL) specification from a C program. The goal of this process is to reduce the complexity of a compiled C program by increasing the efficiency of the instantiated hardware units, as well as to provide a method to remove operations which cannot be performed in hardware and designate them as part of the software pool. The tool produces an optimized GDL description of the input program.

To accomplish this, the tool integrates a C compiler, *lcc*, and a custom parser and dataflow graph processor in order to generate a GDL description of the input code which can be used by the later stages of the compiler. *Lcc* is used to parse C input code and guarantee coverage of the entire C language. The output of *lcc* is in the form of an abstract RTL-like language which is organized into directed acyclic graphs (DAGs). This symbolic language has relatively few operators and can thus be processed easily. Covering all of the RTL-like operations generated by *lcc* ensures total coverage of ANSI-C.

After the symbolic code is parsed and stored in internal data structures, the tool searches all DAGs for the occurrence of operations which cannot be performed in hardware, which have been determined by the developer. These operations may be ones for which the functional units are too resource intensive for the hardware platform, or for which no functional unit has been designed. Operations which cannot be performed in hardware will be executed on the host.

The tool then examines all variables used in the program and performs register allocation. Since the hardware architecture is being specified by the compiler, there is essentially an infinite register

## *CHAPTER 2. BACKGROUND*

pool. As a result, the only variables which are allocated to memory are those which are accessed indirectly, arrays, and globals. Also during this process, all adjacent DAGs containing linear code are merged to form blocks. This process simplifies the control flow analysis which is performed later.

Once the register allocation has been completed, the code is optimized for hardware instantiation. At this point, all constant operand shifts are replaced by constant-shift functional units. In addition, constant operand multiplications are replaced by shift and add sequences.

The next step in the processing of the program is to analyze the control flow information present in the symbolic output. Certain DAGs (conditional and unconditional branches) can be examined along with their target DAGs to get a picture of the structure of the program. In a sense, the structure is being recreated from the symbolic output. However, the processed program is expressed in a simpler form than in the original code.

At this point, the GDL code is generated for each block. In addition, C code for functions in the software pool are placed in another output file for later compilation.

## Chapter 3

### Approach

This chapter presents the methods used to generate the initial graphical description of the input program. In order to provide coverage of the entire ANSI-C language, a commonly available C compiler, *lcc* is used as the front-end to the dataflow analysis tool [8]. *Lcc* is responsible for the parsing and syntax checking of the source language. A mechanism known as the *symbolic* output mode is provided by *lcc* to generate a graphical representation of an input program. Symbolic output is in the form of directed acyclic graphs (DAGs). Individual operations are described graphically and grouped together in *forests* to represent the computation performed by a program. These DAGs are read by the dataflow tool and are processed and optimized in order to generate a GDL description of the program. Within the forest there is a significant amount of explicit control-flow information which must be analyzed or removed prior to generation of GDL code.

The following sections discuss the compilation process from the input C code to the preliminary graphical representation. The symbolic output mode is discussed, and examples are presented. The representation of program structure in the symbolic output is also described. The final part of this chapter describes the Graph Description Language (GDL) which is targeted by the *gdlgen* tool.

## CHAPTER 3. APPROACH

### 3.1 Compilation

There are five main parts of the *lcc* compiler which are responsible for generating a symbolic description of a source program [1]. These five parts are as follows:

**Preprocessor** The preprocessor is responsible for expansion of macros, inclusion of header and source files, and selection of conditionally compiled code [8]. *Lcc* does not contain a built-in preprocessor, but rather, relies on an external preprocessor.

**Lexical Analyzer** The lexical analyzer sometimes known as a scanner breaks the input into discrete tokens [8].

**Parser** The parser uses the structures built by the lexical analyzer to check the program for syntactic correctness. The parser also checks and enforces the type conversion rules of the C language. This stage produces *syntax trees* in which each node represents a basic operation [8].

**DAG Generator** The DAG generator takes the syntax trees created by the parser and converts them into the DAGs which are the fundamental data structure used throughout the rest of the *lcc* compiler and the *gdlgen* tool.

**Emitter** The emitter takes the DAGs and generates the symbolic output of the compiler.

The output of *lcc* forms the input to the dataflow analysis tool.

```

int main()
{
    int i;
    int *j;
    j = &i;
    (*j)++;
    return (i);
}

```

Figure 3.1: Trivial C program.

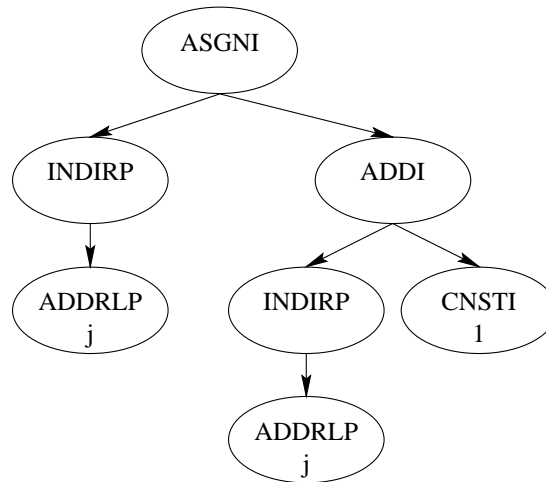


Figure 3.2: DAG generated for  $(*j)++$ ;

### 3.2 Directed Acyclic Graphs

Directed Acyclic Graphs (DAGs) are a natural representation for operations performed in a high-level program. DAGs are hierarchical structures which can easily be processed recursively. The root of any subgraph gets its inputs from its children. This structure lends itself to the hierarchical representation of complex operations. Since each operation in a DAG must not be executed until all of its children have generated results, individual DAGs are the dataflow representation of their

## CHAPTER 3. APPROACH

```

node#3 ADDRPL count=1 j
node#2 INDIRP count=2 #3
node#5 INDIRI count=1 #2
node#6 CNSTI count=1 1
node#4 ADDI count=1 #5 #6
node'1 ASGNI count=0 #2 #4 4 4

```

Figure 3.3: Linearized DAG for  $(*j)++$ ;

Table 3.1: LCC control flow symbolic operations.

<i>Operator</i>	<i>Type Suffix</i>	<i>Operation</i>	<i>Children</i>	<i>Syms</i>
EQ	I F D	Jump if equal	2	1L
GE	I F D	Jump if greater or equal	2	1L
GT	I F D	Jump if greater than	2	1L
LE	I F D	Jump if less or equal	2	1L
LT	I F D	Jump if less than	2	1L
NE	I F D	Jump if not equal	2	1L
JUMP	V	Jump to a label	1	None

corresponding operations. Figure 3.2 shows the DAG generated by *lcc* for the simple indirect increment operation in the C program in Figure 3.1. When *lcc* generates its symbolic output, it linearizes the DAGs by doing a postorder (left-right-root) traversal of each DAG. This process generates a linear list of operations which would preserve instruction dependencies if the operations were to be executed sequentially. Figure 3.3 shows the linearized DAG code that is generated by *lcc* for the DAG in Figure 3.2.

### 3.2.1 Representation of Control Flow Information

In many senses, the symbolic output mode, though expressed graphically is just an abstracted assembly language. This can be seen in the expression of control flow information that is present

Table 3.2: LCC symbolic type suffixes.

<i>Suffix</i>	<i>Type</i>
C	character
S	short
I	integer
U	unsigned
P	pointer
F	float
D	double
B	structure
V	label

in the DAGs. As the compiler processes the input code, it generates labels for DAGs that are referenced by other DAGs. These labels are used by symbolic operations that alter the flow of control, such as branches. These operations are shown in Table 3.1. *Lcc* has both conditional and unconditional branch statements which are used to form the various control flow structures that can be used in the C language. The analysis of control flow information will be more thoroughly discussed in Section 4.8.

### 3.3 Symbolic Operations

*Lcc* has a comprehensive symbolic instruction set. All operations that can be expressed in the source code can be translated into symbolic format. *Lcc* has instructions to perform arithmetic, logical, and type conversion operations, as well as operations which alter the flow of control (i.e. function calls, branches). These symbolic instructions are the fundamental building blocks of the DAGs that are used as the input to *gdlgen*. A complete list of the symbolic datapath operations can be found in Table 3.3. In addition, *lcc* provides a set of data manipulation operations. Both

CHAPTER 3. APPROACH

Table 3.3: LCC datapath symbolic operations.

<i>Operator</i>	<i>Type Suffix</i>	<i>Operation</i>	<i>Children</i>	<i>Syms</i>
CNST	C S I U P F D	Constant	0	1C
CVC	I U	Convert from <i>char</i>	1	None
CVD	I F	Convert from <i>double</i>	1	None
CVF	D	Convert from <i>float</i>	1	None
CVI	C S U D	Convert from <i>int</i>	1	None
CVP	U	Convert from <i>pointer</i>	1	None
CVS	I U	Convert from <i>short</i>	1	None
CVU	C S I P	Convert from <i>unsigned</i>	1	None
NEG	I F D	Negation	1	None
ADD	I U P F D	Addition	2	None
SUB	I U P F D	Subtraction	2	None
MUL	I U F D	Multiplication	2	None
DIV	I U F D	Division	2	None
LSH	I U	Left Shift	2	None
RSH	I U	Right Shift	2	None
MOD	I U	Modulus	2	None
BCOM	U	Bitwise Complement	1	None
BAND	U	Bitwise AND	2	None
BOR	U	Bitwise inclusive OR	2	None
BXOR	U	Bitwise exclusive OR	2	None
ASGN	C S I P F D B	Assignment	2	2S
ARG	I P F D B	Parameter Passing	1	2S
CALL	I F D B V	Function Call	1 or 2	1
RET	I F D	Return Value	1	None
LABEL	V	Label Definition	0	1L

Table 3.4: LCC variable access symbolic operations.

<i>Operator</i>	<i>Type Suffix</i>	<i>Operation</i>	<i>Children</i>	<i>Syms</i>
ADDRF	P	Address of a parameter	0	1V
ADDRG	P	Address of a global	0	1V
ADDRL	P	Address of a local	0	1V
CNST	C S I U P F D	constant	1	1C

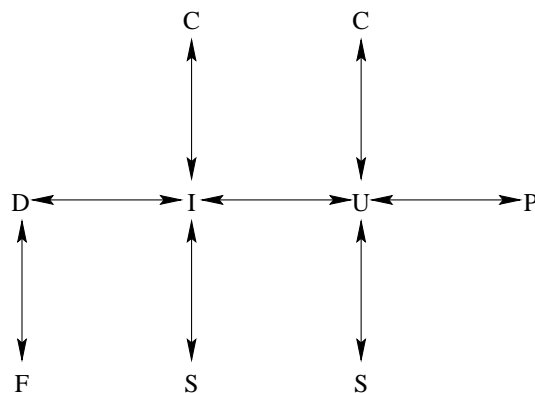


Figure 3.4: Valid type conversions.

an indirection operation, `INDIR`, and an address generation operation, `ADDR`, are provided to load and store values in memory. A list of these operations can be found in Table 3.4.

### 3.3.1 Type Information

Each symbolic operation operates on and produces a specific type of data. The type of data that the operation requires and produces is indicated by the suffix of the operation. Table 3.2 shows the different type suffixes. These conversions can be inserted by *lcc* as a means of accomplishing standard C-style type promotion or can be explicitly specified by the users in the form of a typecast. Figure 3.4 illustrates the valid conversions that *lcc* is capable of performing. All conversions must be made through this hierarchy which is dictated by the ANSI-C language [8]. For example, if an

CHAPTER 3. APPROACH

```
int main()
{
    int i,j;
    j = i & j;
    return (j);
}
```

Figure 3.5: Conversion sample C program.

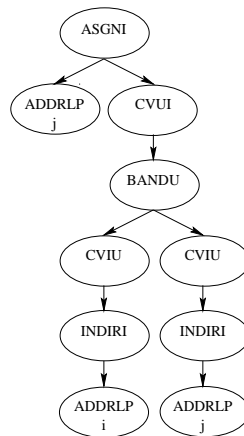


Figure 3.6: Integer to Unsigned DAG.

integer value needs to be converted to a float, it must first be converted to a double, and then to a float. One of the most common conversions that *lcc* performs is a conversion between integer and unsigned values. This conversion is commonly seen when using bitwise logical operations since all symbolic bitwise operations require unsigned inputs and outputs. Figure 3.5 shows a sample program which performs these conversions. Figure 3.6 contains the corresponding DAG for `j = i & j;`.

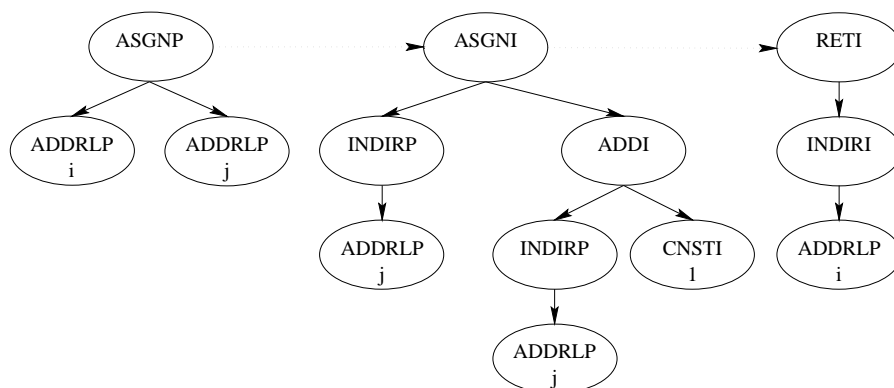


Figure 3.7: Forest for C program in Figure 3.1.

### 3.4 Program Representation

Since each operation in a program is broken up and represented as one or more DAGs in the symbolic code, there must be a mechanism to represent the structure of an entire program. *Lcc* does this by stringing DAGs together in *forests*. Each forest contains the DAGs belonging to a certain function in the original code. A forest is created by linking DAGs together linearly in a list. Figure 3.7 shows a forest for a trivial C program.

### 3.5 Graph Description Language

To represent dataflow graphs in a text-file format, a *graph description language* (GDL) has been developed. To make the graph readable, the GDL has been defined to resemble a high-level language. Loops, conditionals, and function definitions are all available, allowing the programmer to develop an application in GDL if necessary.

Looping structures are provided as part of GDL using the `repeat` and `loop` keywords. Conditional branches may be created with the `decode` and `if/else` keywords. Function declarations

## CHAPTER 3. APPROACH

```
map _f (_a:int,_b:int,_c:int)->ReturnValue:int
{
    ADDI(_a,MULI(_b,_c))->ReturnValue;
} // end _f
```

Figure 3.8: Sample GDL Code.

and function calls are also possible.

The assignment operator, designated by `->`, is used to assign the results of a function to variables. Variables may be local to a function and may store temporary results, or they may refer to the input and output parameters of the function. Local variables need not be specified explicitly, and the meaning of a variable name changes as the graph file is read sequentially to refer to the last occurrence of assignment to the name. While this is not traditional dataflow, it allows variable names to be reused, providing a more friendly environment to the programmer reading the code. Regardless, the code is translated into its dataflow equivalent. An example of this input language is given in Figure 3.8. Inclusion of branching and looping structures in the graph language allows the compiler to create a more efficient implementation of a given graph, since it doesn't resort to purely data- or demand-driven techniques [22]. In pure data-driven execution, both sides of a conditional are evaluated and the result is multiplexed by the condition, thus wasting computing resources on the unused branch. In pure demand-driven execution, neither side of a conditional is executed until the condition is evaluated, determining which side will execute. While the demand-driven approach does not perform needless computation, neither does it allow for speculative execution of the two branches. Since the control flow aspects of the conditional remain in the graph language, the compiler may schedule speculative execution of a branch which will terminate if the condition

### *CHAPTER 3. APPROACH*

determines that it is unneeded. With explicit control flow aspects, the compiler can avoid scheduling such operations if they are not warranted due to the value of the conditional [19]. Furthermore, another deviation from pure dataflow is that certain operations specified in the graph may have side-effects, such as memory write and I/O operations.

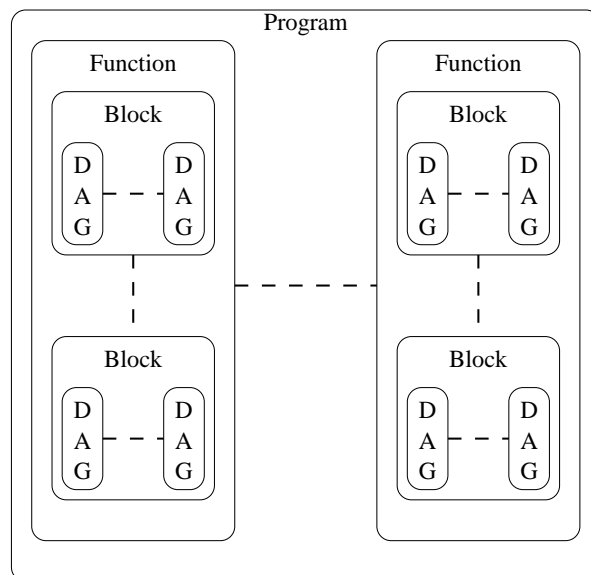
## Chapter 4

### GDLGEN

Once the source code has been processed by *lcc*, the compilation process moves on to the dataflow analysis stage. Dataflow analysis is performed by a custom tool, *gdlgen*, which is specifically designed to work with *lcc* to generate input code to the *SAS* scheduler of the *PRISM* compiler. The *gdlgen* tool analyzes and optimizes the dataflow description generated by *lcc* in order to provide a Graph Description Language (GDL) description of the input program to the *SAS* tool.

The GDL code is emitted from the *gdlgen* tool through a series of steps. The first step is the parsing of the symbolic input code. The parser creates a hierarchical representation of the input code which is used and modified throughout the program. Once the program has been parsed, the tool examines the code and looks for any occurrences of symbolic operations which cannot be mapped to hardware. Any graph containing these operations is automatically moved to the software pool. The tool then proceeds to perform register allocation on the program. After register allocation, the tool performs hardware-specific optimizations. Optimizations will be more thoroughly discussed in Chapter 5. At this point, the tool analyzes the control-flow information present in the optimized program, and then emits the equivalent GDL code.

This chapter describes in detail the operation of the *gdlgen* tool. First, the general model of program representation is described, then the implementation methodology is described. Examples

Figure 4.1: *Gdlgen* program structural representation.

of the translation from C to GDL are given later in the chapter. Throughout this chapter, *gdlgen* program objects such as the `Function` object will be referred to in a typewriter style font. When an actual source code function is being referred to, it will be in a normal font.

#### 4.1 Program Representation

*Gdlgen* uses a hierarchical representation of its input code. A single input file corresponds to a single `Program` object. Each `Program` object contains an arbitrary number of `Function` objects. Each `Function` object in turn contains an arbitrary number of `Block` objects. A `Block` object can contain an arbitrary number of DAG objects. Although *lcc* considers a DAG a fundamental unit of a program, *gdlgen* considers a `Block` to be the fundamental unit of a program since a `Block` is a group of DAG objects which are adjacent and linear. The program structure is shown in Figure 4.1.

```
int main()
{
    int j;
    float l;
    l = j * 5.0;
    return(j);
}
```

Figure 4.2: A simple C program.

## 4.2 Parsing

Once *lcc* has written the symbolic output to a file, the *gdlgen* tool must parse the symbolic code to analyze it. *Gdlgen* implements a fairly simple template-based parser since the symbolic code is always correctly formatted, and is always in a predictable format. The main job of the parser is to create a complete and accurate hierarchical representation of the input code. It does this by making several passes through the input file to build the program hierarchically. A sample C program is shown in Figure 4.2, and its corresponding symbolic output code is shown in Figure 4.4.

### 4.2.1 Global Symbols and Variables

The first thing that the parser does is to examine the entire symbolic file for global symbols and variables. Global symbols are generated by *lcc* when it encounters some constants which cannot be placed in a DAG. For example, *lcc* creates global symbols for all `double` constants. Any time a user declares a variable globally, that variable is declared globally in the symbolic output. Since both global variables and global symbols reside outside the boundaries of any function, they are placed directly into a `Program` object. Global symbols are then added to the global symbol list object, and

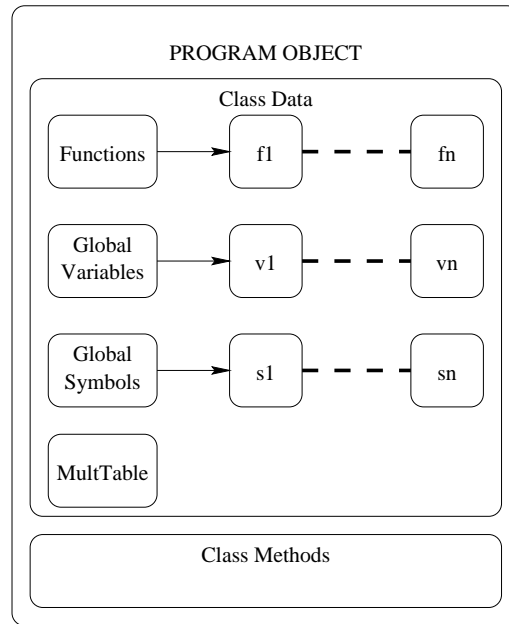


Figure 4.3: Structure of the Program object.

global variables are added to the global variable list object. The structure of the Program object can be seen in Figure 4.3. Global symbols are discussed further in Section 4.3.

### 4.2.2 Functions

After the parser has read all global symbols and variables, it then proceeds to read all of the function declarations. Each function declaration is parsed into a Function object, which contains fields for both the parameters and the local variables that are declared within the function. Once the declaration has been parsed, the new Function object is added to the function list in the Program object. Note that since ANSI-C does not allow nested functions, every function declared in a program will have an entry in this list.

## CHAPTER 4. GDLGEN

```
export main
segment text
function main type=int function(void) class=auto scope=GLOBAL ref=0
local j type=int class=auto scope=LOCAL offset=0 ref=2000
local l type=float class=auto scope=LOCAL offset=4 ref=1000
maxoffset=8
node#2 ADDR LP count=1 l
node#6 ADDR GP count=1 2
node#5 INDIR count=1 #6
node#9 ADDR LP count=1 j
node#8 INDIR count=1 #9
node#7 CVID count=1 #8
node#4 MUL D count=1 #5 #7
node#3 CVDF count=1 #4
node'1 ASGNF count=0 #2 #3 4 4
node#3 ADDR LP count=1 j
node#2 INDIR count=1 #3
node'1 RETI count=0 #2
1:
end main
segment lit
global 2 type=double class=static scope=GLOBAL ref=1000
defconst double 5.000000000000000000e+00
.end
```

Figure 4.4: Symbolic output for C program in Figure 4.2.

## CHAPTER 4. GDLGEN

### 4.2.3 Code

Once the entire *structure* of the program has been read, the parser returns and starts examining the forests which represent the code for individual functions. Since the declarations for **Function** objects have already been placed in the **Program** object, the parser simply traverses the list of **Function** objects and parses the code for each function.

When a new function is found, the parser retrieves the **Function** object for the new function by getting the next entry in the function list. The parser then proceeds to create a new **Block** object and parse the code. At this stage, each **Block** contains only one DAG for simplicity. **Block** objects are merged later in the process to simplify the control-flow analysis.

### 4.3 Global Symbol Resolution

Once the entire program has been parsed, the tool searches the code for any reference to a global symbol that can be placed in a DAG. Specifically, the tool is looking for references to constants that have been declared as global symbols as opposed to **CNST** (constant) nodes in the DAGs. Since GDL can handle constants of any type, it is simpler to move the constant into the DAG than to resolve it as the GDL code is being emitted. If any of these references are found, the value of the associated global symbol gets inserted into the DAG, and the fetch operation is removed from the DAG.

```
int main()
{
    float a,b;
    float c;
    a = b*c;
    b = a/b;
    if (b < 10.)
        c = a/b;
    return 0;
}
```

Figure 4.5: A C program which will demonstrate the hardware/software allocation.

#### 4.4 Hardware and Software Pool Allocation

There may be some functional units which are not available either because of limited use or because a functional unit is too large for a given resource pool. As a result, there may be some operations which cannot be implemented in hardware. At this level, the definition of operations which cannot be implemented in hardware is left to the user, or to an external tool which evaluates both the library and the architecture description for resource conflicts. To handle these cases, the *gdlgen* tool performs preliminary partitioning of input programs into hardware and software pools based on these user defined resource or library conflicts. This mechanism does not take into account any resource utilization issues due to scheduling and partitioning, rather, it just tries to keep the compiler from attempting to map unmappable graphs into hardware. All graphs which are moved to the software pool will be executed on the host processor.

The tool accomplishes this by reading a configuration file, `pool.conf`, and searching the input program for occurrences of operations that appear in this file. When such an occurrence is found,

## CHAPTER 4. GDLGEN

```
map _main ()->ReturnValue:int
{
//Local variable declarations.
0:float->_c;
0:float->_a;
0:float->_b;

    funcGDLGEN_0(_a,_b,_c)->(_a:float,_b:float,_c:float);
    if(NOT(GED(CVFD(_b),1.0000000000000000e+01:double)))
    {
        funcGDLGEN_1(_c,_a,_b)->(_c:float,_a:float,_b:float);
    }
    0:int->ReturnValue;
} // end _main
```

Figure 4.6: The GDL translation of the C code in Figure 4.5 showing generated function calls.

*gdlgen* moves the block in which the operation is found from the hardware pool to the software pool. A new temporary function call is also generated to represent the call into the software pool. The function call mechanism is already in place and available in GDL, and handles all communication between the calling and called functions. Figure 4.5 shows a C program which demonstrates the hardware software pool allocation. In this example, the operation DIVF is not mappable to hardware. As a result, the two placeholder function calls `funcGDLGEN_0` and `funcGDLGEN_1` are inserted. This is shown in Figure 4.6. This allows large portions of functions that contain unmappable operations to be executed in hardware, while only executing the necessary graphs in software on the host processor.

## 4.5 Label Merging

In order to eliminate redundant labels which were generated by *lcc*, the *gdlgen* tool merges all adjacent labels. This is also done to simplify the control-flow analysis. The code is searched for all references to redundant labels, which are replaced by references to the new merged label.

## 4.6 Register and Memory Allocation

Since the compiler has the freedom to specify any hardware architecture that is best suited to perform the computations specified by the program, it also has the freedom to allocate as many registers as required to perform the program. The symbolic code assumes that all variables will be accessed as if they were in memory. The *gdlgen* tool must therefore make use of any information available to eliminate as many memory accesses as possible, since memory accesses are extremely inefficient compared to register accesses.

### 4.6.1 Registered Variables

*Gdlgen* examines every variable in a function and looks for any instance where it is accessed indirectly, i.e. through a pointer. When it finds a variable which is *never* accessed indirectly, it removes all memory-fetch operations performed on that variable. That variable is then marked as a REGISTER variable throughout the entire function. This search is performed only within a function on its local variables and parameters. Note that this search is not performed on global variables. It is assumed that global variables will be accessed from many different graphs, and will therefore be left in memory.

```
int f(i,j) int i, j;
{
    return i+j;
}
void main(void)
{
    int i;
    int j;
    int *k;

    k = &j;
    (*k) = 5;
    i = f(j,(*k));
}
```

Figure 4.7: Example C program containing pointer manipulations.

It is important to note that since this register allocation process is performed on parameters which are passed to a function, that the changes made are strictly local to the function. For example, if a parameter is passed in by value (copy) and is subsequently accessed indirectly, only the local copy of the variable is placed in memory. The tool also provides startup code for any such variables which must be moved into memory at the beginning of a function. The startup code can be seen in Figure 4.8, where `_j` is initialized using an `MWRITE` function.

### 4.6.2 Memory Variables

If a variable is accessed indirectly, the tool will leave graphs containing the variable intact. When the GDL emitter traverses the graph, a memory read or write operation will be generated. Since the tool has very limited knowledge of the architecture of the CCM being targeted, it has no way of being able to assign addresses to variables. Instead, when a memory fetch is generated, the

## CHAPTER 4. GDLGEN

```
map _f (_i:int,_j:int)->ReturnValue:int
{
    ADDI(_i,_j)->ReturnValue;
} // end _f

map _main ()->ReturnValue:void
{
    //Local variable declarations.
    0:void->MemoryDep;
    0:int->_i;
    MWRITE(0:int,@_j:int,MemoryDep)->MemoryDep
    0:pointer->_k;

    @_j:int->_k;
    5:int->_k;
    _f(MREAD(@_j:int,MemoryDep),MREAD(_k,MemoryDep))->_i;
} // end _main
```

Figure 4.8: GDL code showing pointer manipulations.

tool generates a special tag that indicates that a *linker* must allocate memory and assign a base address. A variable that needs to be allocated to memory appears in the GDL code as follows:

```
@_j:int
```

This means that variable `j` must be allocated to memory, and enough space must be allocated for an `int`. Additionally, if a memory operation is present in a function, a dependency signal, `MemoryDep` is placed in the function and is threaded as an argument through all memory access functions. This is to guarantee that memory accesses occur in order. Figure 4.8 shows the GDL translation containing pointer manipulations of the C program in Figure 4.7.

## 4.7 Block Merging

After the register and memory allocation has taken place, the tool then begins to search all of the blocks of code that have been parsed. The purpose of this search is to merge all adjacent blocks that contain linear code. Linear code in this application means that the block is not a label, and does not alter the flow of control. As stated in Section 4.2, each DAG is read into a separate `Block` object. In this stage, adjacent `Block` objects are merged if and only if they both contain linear DAG objects. The reason that this merging is performed is to simplify the control-flow analysis that is discussed in Section 4.8. This merge decreases the number of code blocks that the control-flow analyzer must consider.

## 4.8 Analysis of Control-Flow Information

At this point, the tool is ready to perform the control-flow analysis, which is later used by the GDL Emitter (Section 4.11). When looking through the code, there are four different types of branches that can occur. Unconditional branches forward and backward, and conditional branches forward and backward. The simplest cases are the conditional branches.

### 4.8.1 Conditional Branches

Conditional branches represent the simplest case in the control-flow analysis. They are simplest because they can be interpreted directly. A conditional branch forward always represents an `if` statement in GDL. A conditional branch backward always represents some kind of loop structure in GDL.

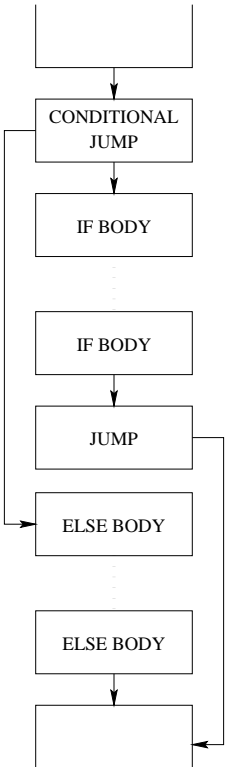


Figure 4.9: Program block structure for *if-else* construct.

### Conditional Branches Forward

In *lcc*, `if` statements are compiled into structures such as the one shown in Figure 4.9. Since the first conditional branch branches around a section of code, it is the inverse of the condition that executes the `if` body code. The GDL Emitter handles this case by automatically inverting the condition of `if` statements. When a conditional branch forward is encountered, *gdlgen* tags it as an `if` statement and then finds the target of the jump. The target block is examined and is tagged as an `if end` block. This procedure is capable of handling multiple levels of `if` statements. In fact, since there is no facility for an `else if` structure in GDL, `else if` structures are implemented as a nested `if` within an `else` clause.

### Conditional Branches Backward

Conditional branches backward are slightly different than the conditional branches forward. Since backward branches imply a loop construct, they can not skip over blocks of code (unless of course they contain nested `ifs`). Because of this, when a conditional branch backward is encountered, it is followed in order to examine all of the DAGs. When a conditional branch backward is encountered, it is tagged as a `loop` statement, and its target is tagged as a `begin loop` block. Figure 4.10 shows the block structure of a `while` loop. Again, multiple levels of nested loops can be handled.

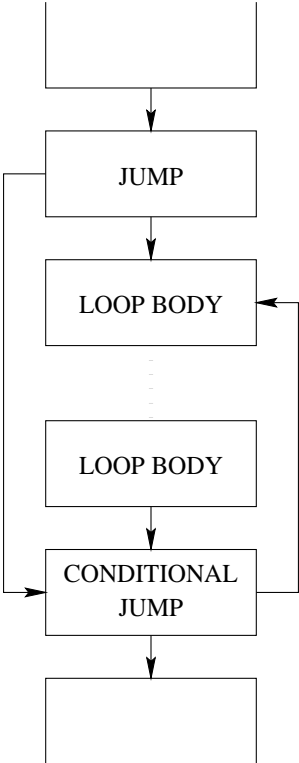


Figure 4.10: Program block structure for *while* loop.

## 4.9 Unconditional Branches

Unconditional branches in the symbolic code have no direct translation into GDL code. Because of this, any unconditional branches present in the symbolic code must correspond to one of the control-flow constructs that can be represented in GDL. Unconditional branches which can be handled in GDL must only be used to “restore” the flow of control rather than to alter it. Unconditional branches are common blocks that represent the end of `if`, `else` and `loop` blocks in GDL. The context of the branch determines what type of GDL construct it represents. For example if an unconditional branch forward has a target that is a conditional branch backward, the unconditional branch is the beginning of a `loop` block in GDL as can be seen in Figure 4.10. If an unconditional branch is executed as a result of a `goto` statement in the C source, this branch will have no equivalent GDL construct.

## 4.10 The Switch-Case Construct

Since *lcc* generates a block pattern for switch-case constructs that is different from any other construct, a special function was designed to recognize them. This function also rearranges the blocks so that the construct appears to be an `if-else` chain. There are some limitations to this, however. Currently, multiple case labels and cases without `break` statements are not handled. Figure 4.10 shows the block structure of a case statement. Note that this structure is transformed *before* the control-flow analysis into the block structure for an `if-else` construct as shown in Figure 4.9.

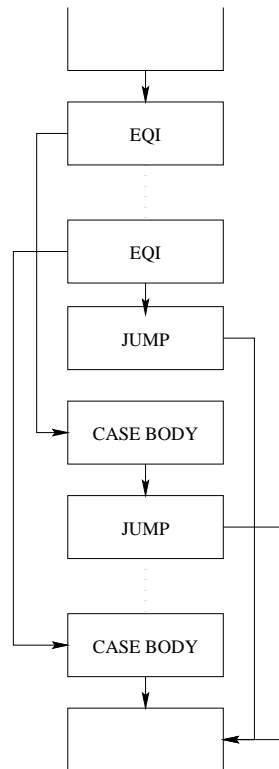


Figure 4.11: Program block structure for a switch-case statement.

## CHAPTER 4. GDLGEN

### 4.11 GDL Emission

Once the control flow analysis has been completed, the GDL Emitter takes over and, based on the information supplied by the control flow process, generates GDL code. Since the control flow analysis has tagged all blocks of code with the type of GDL structure that they represent, the GDL Emitter just examines these tags and produces formatted GDL code.

### 4.12 Modifications to *lcc*

One change that was made to *lcc* in order to simplify the analysis was to prevent it from generating references outside of the current DAG. Before the modification, *lcc* would generate references to previous DAGs in order to keep from repeating a code sequence. In order to prevent this from happening, a small modification was made to the DAG generator to generate complete DAGs. To date, this is the only modification that has been made to *lcc*.

# Chapter 5

## Optimization Strategy

Optimizations in a hardware compiler require a different approach than optimizations in a traditional software compiler. These optimizations are generally focused on the instantiation of efficient (area and time) hardware functional units. Since traditional general-purpose microprocessors are required to be complete to execute any type of program, they must have general-purpose functional units (i.e. barrel shifters, multipliers). These general-purpose functional units are capable of handling any variation of the operation they were designed to perform. As a result, these units are not as efficient as they could be had they been designed to perform a specific function.

In order to make the specific instantiation of a given program most efficient in hardware, several different optimizations have been implemented. The optimization relied on most heavily is strength reduction—the process of decomposing complex operations into a series of simpler operations [12]. This approach allows both time- and area-efficient implementations of certain functional units to be used by the scheduler. The operations given the most attention are shifting and multiplication by constants. It should be noted that *lcc* performs some strength-reducing optimizations itself, specifically when a multiplication or division operation is by a power of two.

This chapter describes the methods of optimization used in the *gdlgen* tool. These optimizations all have a common goal: to make the instantiated hardware as efficient as possible.

```

int func(a, b) int a, b;
{
    a = a+2*b;
    return a;
}

int main(a,b,c,d) int a,b,c,d;
{
    if (d < c)
        d = func(a,b);
    else
        d = func(a*34,b);
    return d;
}

```

Figure 5.1: Multiplication chain example C program.

## 5.1 Multiplication Strength Reduction

Implementing full multipliers is expensive in terms of logic given the resources of each FPGA. In addition, approximately 91% of multiplication operations include constant operands [16]. In order to eliminate the need for providing full multipliers for each multiply operation in a graph, the analysis tool detects all instances of multiplication by a constant. These instances are then transformed into a series of shifts and adds through the use of simple chain rules [15]. This allows the use of simple, area-efficient functional units (shifters and adder/subtractors) to implement these multiplications.

### 5.1.1 Multiplication Chains

Since the generation of these chains is computationally intensive, the compiler relies on an existing database of all possible multiplications from 2 to 65,536. The *length* of a particular chain

## CHAPTER 5. OPTIMIZATION STRATEGY

```
map _func (_a:int,_b:int)->ReturnValue:int
{
    _a->_a;
    ADDI(_a,LSHI1(_b))->_a;
    _a->ReturnValue;
} // end _func

map _main (_a:int,_b:int,_c:int,_d:int)->ReturnValue:int
{
    if(NOT(GEI(_d,_c)))
    {
        _func(_a,_b)->_d;
    }
    else
    {
        _func(LSHI1(ADDI(_a,LSHI4(_a))),_b)->_d;
    }
    _d->ReturnValue;
} // end _main
```

Figure 5.2: Multiplication chain GDL code example

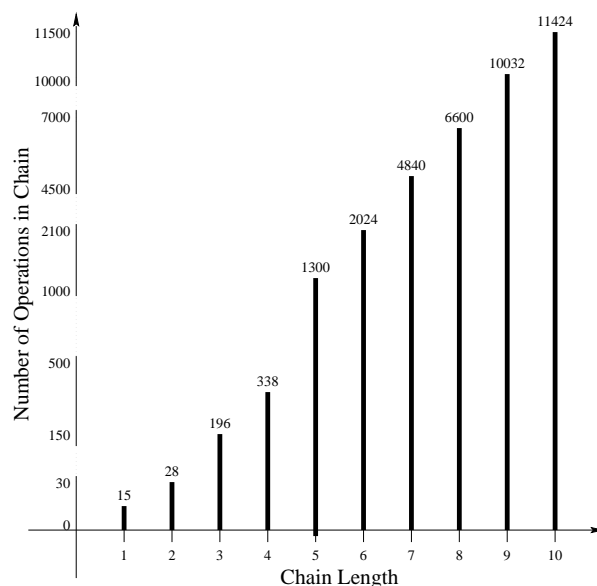


Figure 5.3: Histogram of chain lengths up to 10.

is defined as the number of operations that are required to compute its result. The length of all chains between 2 and 65,536 is less than 20. Practically though, instantiating 20 functional units to save the instantiation of a single multiplier is probably not a reasonable trade, even when the shift units that are instantiated take up no resources. As a result of this, a cap of 10 was placed on the length of the chains. This provides a reasonable tradeoff between the instantiation of the multiplier and the instantiation of between 3 and 7 adders.

Out of 65,536 constants, 36,800 or 56% of them have equivalent chains of length less than or equal to 10. Figure 5.3 shows a histogram of the chain lengths up to 10.

### 5.1.2 Replacement

When the *gdlgen* tool finds a multiplication by a constant, it searches the database to see if there is a chain for that constant. If there is, the new chain is substituted for the old multiplication,

## CHAPTER 5. OPTIMIZATION STRATEGY

and will then appear in the GDL code output. An example of this optimization can be seen in Figures 5.1 and 5.2. The multiplication by 34 in the call `d = func(a*34);` gets converted to an equivalent chain.

### 5.2 Shift Strength Reduction

As with multipliers, variable shifters are expensive to implement. In addition, since very few variable-amount shifts are present in common HLL programs, any constant shifts found in the input are implemented as constant shift units.

The complexity in a barrel shifter is derived from the need to handle different shift amounts all within the same amount of time. On the other hand, a serial shifter, while simpler, may require a significant number of cycles to complete. Because the shift amount in most shift operations is known at compile time, there is no need to incur either the resource penalty required to instantiate a barrel shifter, or the time penalty required to instantiate a serial shifter. Constant shift units require no logic resources, they consume only routing resources. Because of this lack of logic, the constant shift units require basically no time to complete their task.

As a result, the *gdlgen* tool detects all shift operations with a constant argument and replaces these shifts with constant-shift units. These constant-shift units are now simply a routing function mapping bits of the input to different bits of the output.

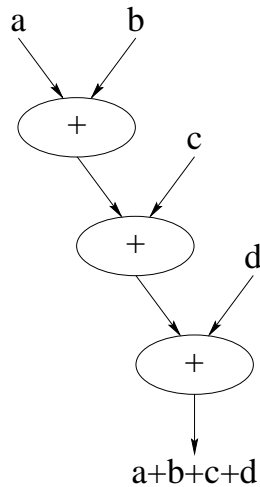


Figure 5.4: An unbalanced dataflow graph.

### 5.3 Balancing of Arithmetic DAGs

An optimization that would benefit the *PRISM* compiler would be the balancing of DAGs representing arithmetic operations. Because *lcc* tends to optimize very little, there are many DAGs which are not optimal, that is they do not exploit the parallelism inherent in the operation they represent. As an example, consider the computation  $a+b+c+d$ . This simple sum can be expressed in several ways. Since input is parsed from left to right, a logical representation of the operation is  $(a+(b+(c+d)))$ . This implies that the sum  $c+d$  is added to  $b$ , and that this partial sum is then added to  $a$  to get the complete sum. In GDL, this sum would be expressed as:

```
ADDI(a, ADDI(b, ADDI(c, d)))
```

which is clearly not optimal since the sums  $a+b$  and  $c+d$  could be carried out in parallel. Figure 5.4 shows the dataflow representation of this sum.

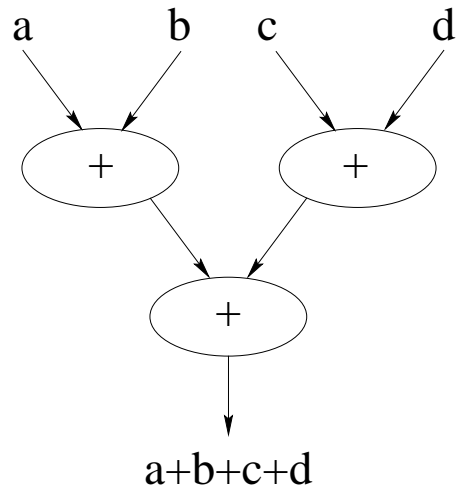


Figure 5.5: An equivalent balanced dataflow graph.

This operation could be balanced by rearranging the order of the operations. The desired operation calculates the partial sums  $a+b$  and  $c+d$  and then adds them together to get the complete sum. In GDL this sum would be expressed as:

```
ADDI(ADDI(a,b),ADDI(c,d))
```

which expresses the parallelism inherent in the graph for the operation. Figure 5.5 shows the dataflow representation of this new sum. Although this optimization has not yet been implemented, *gdlgen* has been designed to accommodate it. The `Program` object already has a method `BalanceDAGs`. This will be discussed further in Chapter 7.

# Chapter 6

## Experimental Results

The dataflow analysis of the input C programs is an important step in the compilation of C programs to hardware descriptions. The next step in the process is accomplished by the *SAS* scheduler. This tool partitions the dataflow graphs which represent the program over the different resource pools which are available within the target architecture. In addition, it provides a mechanism to share functional units temporally by creating *schedules*.

This chapter shows examples of the translation of a C program to a schedule. There are two examples, one which is a relatively simple example to illustrate the dataflow analysis and scheduling process, the other is more complex and shows the tools handling a larger input program.

### 6.1 A Simple Example

Figure 6.1 shows a simple C program which is used as the input to the compiler. There are several important features in this C program which illustrate properties of the compiler. First, the analysis of `if-else` blocks is shown. Second, the function-call mechanism is shown. Third, the reduction of a multiplication by a constant to a series of shifts and adds is shown.

The GDL code in Figure 6.2 is the translation produced by *gdlgen* of the C code in Figure 6.1. Note that every operation in the GDL code is essentially a function call, and that its type is

## CHAPTER 6. EXPERIMENTAL RESULTS

```
int func(a, b) int a, b;
{
    a = a+2*b;
    return a;
}

int main(a,b,c,d) int a,b,c,d;
{
    if (d < c)
        d = func(a,b);
    else
        d = func(a*34,b);
    return d;
}
```

Figure 6.1: Simple C code example.

indicated by the suffix of the operator. Also note the reduction of the multiplication `a*34` to a series of shifts and adds.

Along the left side of the schedule is a list of all of the instantiated functional units. The  $x$  axis represents time, and the lines between functional units represent datapaths or data dependencies. The generated schedule is contained entirely within a single resource pool (a single XC4010 FPGA).

Together with an architecture description, the GDL code in Figure 6.2 is given to the *SAS* tool for generation of a hardware description of the program. *SAS* generates the schedule which is shown in Figure 6.3.

### 6.2 A More Complex Example

The C program in Figure 6.4 is significantly more complicated than the previous example. There are several reasons for this. First, the program makes use of both loop and conditional structures.

CHAPTER 6. EXPERIMENTAL RESULTS

```
map _func (_a:int,_b:int)->ReturnValue:int
{
  _a->_a;
  ADDI(_a,LSHI1(_b))->_a;
  _a->ReturnValue;
} // end _func

map _main (_a:int,_b:int,_c:int,_d:int)->ReturnValue:int
{
  if(NOT(GEI(_d,_c)))
  {
    _func(_a,_b)->_d;
  }
  else
  {
    _func(LSHI1(ADDI(_a,LSHI4(_a))),_b)->_d;
  }
  _d->ReturnValue;
} // end _main
```

Figure 6.2: Simple GDL code for C program in Figure 6.1.

CHAPTER 6. EXPERIMENTAL RESULTS

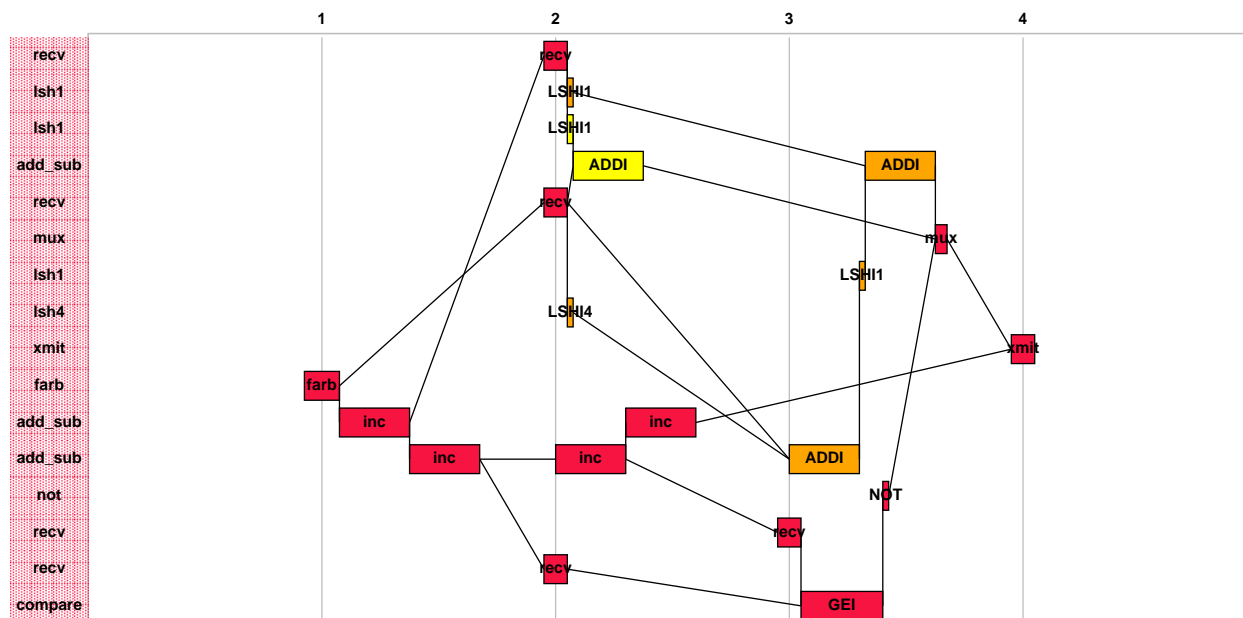


Figure 6.3: A schedule for the GDL code in Figure 6.2.

Second, the program generates extensive type conversion operations. Third, the program uses a “compound” loop statement of the form `while(a && b)` representing the boolean AND of the two conditions. This construct generates a nested `if` in GDL which assigns a 0 or 1 value to a temporary signal which is subsequently used in the loop condition.

The GDL code for this example is shown in Figure 6.5. Again, this GDL code is given to the *SAS* tool for scheduling and partitioning. The schedule generated by *SAS* is shown in Figure 6.6. The schedule generated occupies three resource pools (each resource pool represents a single XC4010 FPGA) primarily due to the size of the floating point functional units. Different resource pools are denoted by different shading along the *y* axis.

## CHAPTER 6. EXPERIMENTAL RESULTS

```
void mandel(float llr,float lli,float urr,float uri)
{
    unsigned char r,i,c,t;
    float zr,zi,cr,ci,tri,trr,tii;

    for(i=0; i<100; i++)
    {
        ci=lli+(uri-lli)/100.0*i;
        for(r=0; r<100; r++)
        {
            cr=llr+(urr-llr)/100.0*r;
            zr=cr; zi=ci; c=1;
            do
            {
                trr=zr*zr;
                tii=zi*zi;
                t=(trr+tii)<2.0;
                if(t)
                {
                    tri=zr*zi;
                    zr=trr-tii+cr;
                    zi=2*tri+ci;
                    c++;
                }
            } while(t && c<255);
            plot(r,i,c);
        }
    }
}

int main()
{
    mandel(-1.5,1.25,1,-1.25);
    return 0;
}
```

Figure 6.4: A C program to calculate a Mandelbrot set.

## CHAPTER 6. EXPERIMENTAL RESULTS

```

map _mandel (_llr:float,_lli:float,_urr:float,_uri:float)->ReturnValue:void
{
//Local variable declarations.
0:int->_i5; 0:uchar->_i; 0:uchar->_r; 0:float->_ci;
0:float->_cr; 0:float->_tri; 0:uchar->_c; 0:float->_tii;
0:float->_trr; 0:uchar->_t; 0:float->_zi; 0:float->_zr;
0:char->_i;
  repeat
  {
    loop(LTI(CVUI(CVCU(_i)),100:int));
    CVDF(ADDD(CVFD(_lli),MULD(DIVD(CVFD(SUBF(_uri,_lli)), 1.0000000000000000e+02:double),
      ADDD(MULD(2.0000000000000000e+00:double, CVID(CVUI(RSHU1(CVCU(_i))))),
        CVID(CVUI(BANDU(CVCU(_i),1:uint)))))))->_ci;
    0:char->_r;
    repeat
    {
      loop(LTI(CVUI(CVCU(_r)),100:int));
      CVDF(ADDD(CVFD(_llr),MULD(DIVD(CVFD(SUBF(_urr,_llr)), 1.0000000000000000e+02:double),
        ADDD(MULD(2.0000000000000000e+00:double, CVID(CVUI(RSHU1(CVCU(_r))))),
          CVID(CVUI(BANDU(CVCU(_r),1:uint)))))))->_cr;
      _cr->_zr; _ci->_zi; 1:char->_c;
      repeat
      {
        MULF(_zr,_zr)->_trr; MULF(_zi,_zi)->_tii;
        if(NOT(GED(CVFD(ADDF(_trr,_tii)),2.0000000000000000e+00:double)))
        {
          1:int->_i5;
        }
        else
        {
          0:int->_i5;
        }
        CVUC(CVIU(_i5))->_t;
        if(NOT(EQI(CVUI(CVCU(_t)),0:int)))
        {
          MULF(_zr,_zi)->_tri; ADDF(SUBF(_trr,_tii),_cr)->_zr;
          ADDF(MULF(2.00000000e+00:float,_tri),_ci)->_zi; CVUC(CVIU(INCI(CVUI(CVCU(_c)))))->_c;
        }
        EQI(CVUI(CVCU(_t)),0:int)->LoopTmp0;
        if(NOT(LoopTmp0))
        {
          LTI(CVUI(CVCU(_c)),255:int)->LoopTmp0;
        }
        loop(LoopTmp0);
      }
      CVUC(CVIU(INCI(CVUI(CVCU(_r)))))->_r;
    }
    CVUC(CVIU(INCI(CVUI(CVCU(_i)))))->_i;
  }
} // end _mandel
map _main ()->ReturnValue:int
{
  _mandel(-1.50000000e+00:float, 1.25000000e+00:float, 1.00000000e+00:float, -1.25000000e+00:float);
  0:int->ReturnValue;
} // end _main

```

Figure 6.5: A GDL translation of the Mandelbrot program in Figure 6.4.



## Chapter 7

### *GDLGEN* Tool Organization

The *GDLGEN* tool is written entirely in C++. Because of this, the structure of objects which are used by the tool can be easily thought of as representing typical user programs. C++ provides not only the language for program development, but also the general methodology of *object oriented* programming (OOP) [21].

This chapter describes the organization of the *gdlgen* tool itself. The structural representation of user programs is described, as is the algorithmic flow of a complete analysis of a symbolic file. Figure 7.1 shows the organization of the tool at the highest level.

#### 7.1 General Structure

At the highest level, a user program is viewed as a collection of resource pools corresponding to the hardware and software resources of the CCM and the host platform. Resource pools are represented by **Program** objects. The hardware resource pool capabilities are defined by the user to account for limitations of the CCM platform or library deficiencies. These capabilities are the only limitation placed on the pools. In all other senses, the *gdlgen* tool considers the pools to be infinite, since it is an architecture independent tool.

The resource pools are composed of arbitrary numbers of **Function** objects which represent

## CHAPTER 7. GDLGEN TOOL ORGANIZATION

both the actual functions that were declared in the input C program, and functions which were generated by *gdlgen* to handle the hardware resource pool limitations. Each **Function** is composed of multiple **Block** objects. The **Block** objects are collections of DAGs. Each **Block** contains a maximal number of consecutive linear DAGs. A linear DAG is defined as a DAG which executes without altering the flow of control. A **DAG** is a collection of individual symbolic instructions grouped together in a dataflow representation of a single atomic operation.

### 7.2 Program Flow

When the *gdlgen* tool is launched, it processes as many symbolic files as are provided on the command line. The flow described here is for a single symbolic file. The process is repeated for each additional symbolic file which is processed.

When the program starts, the command line arguments are parsed, and any flags are set. Currently, *gdlgen* supports two command line flags: `-v` to put the tool into *verbose* mode and `-h` to stop the tool from performing the hardware/software pool allocation. This step is only performed when the tool is started, as the flags apply to the processing of all symbolic files appearing on the command line.

The first step in processing a symbolic file is to parse in the description which has been generated by *lcc*. To perform the parsing, there are two objects used: a `cFileSymbolic` object which abstracts the standard file operations, and provides the `Parse` object with a simple interface to a symbolic file. The parser simply creates a `cFileSymbolic` object and calls its `GetNextToken` method. The `GetNextToken` returns an integer corresponding to the type of the token, as well as

## CHAPTER 7. GDLGEN TOOL ORGANIZATION

a string containing the token itself. The parser reads through the symbolic file several times. The first pass establishes the structure of the program and parses all of the function declarations. The second pass parses in the code for all of the functions, and the third pass reads in the information on global symbols (constants and variables).

Once the program has been parsed in, it is all in a single hardware pool, `Ph`. The hardware pool then calls the `Program` method `ResolveGlobals` to replace any references to global symbols with their actual value so that they represent either simple inlined constants or normal variable accesses.

After the global references have been resolved, the tool attempts to form the `Block` objects from the existing `DAG` objects by finding adjacent linear `DAGs` and merging them. This process is all performed by the the `Program` method `Blockify`.

Once the `Block` objects have been formed, the next step is to examine the hardware pool for any operations that cannot be implemented in hardware. The tool does this by instantiating a `Pooler` object that can operate on both the hardware and software pools. This object is created separately because it must be separate from any resource pool.

The program then attempts to merge any labels that are adjacent to each other to form a single label. This is done by calling the `Program` method `MergeLabels`. Duplicate labels are merged by this method, and all of the references to duplicate labels are replaced by references to the label which remains.

The next step is to perform register allocation. This is accomplished by calling the `Program` method `AllocateRegisters`. This method performs resource allocation on a resource pool. The

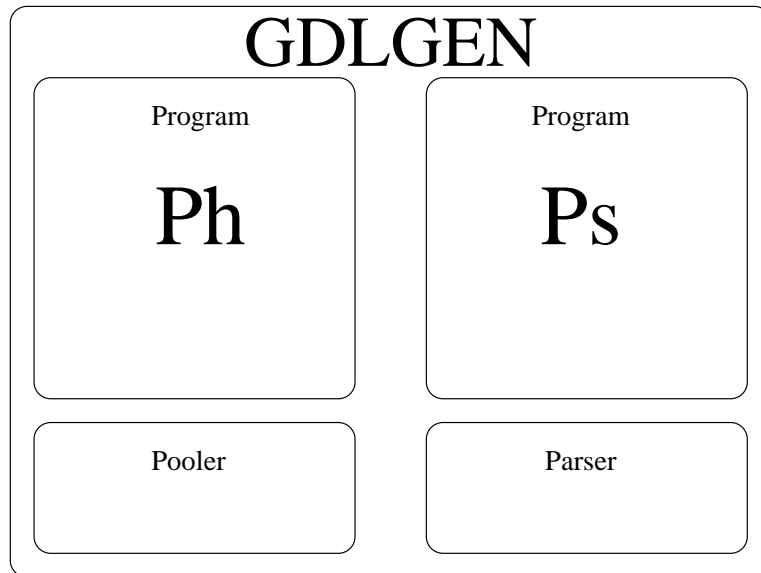


Figure 7.1: The structure of the *gdlgen* tool.

`AllocateRegisters` method uses many other methods in objects below the `Program` object. The `AllocateRegisters` method iterates over all `Function` objects in a pool and calls each `Function` object's `AllocateRegisters` method. The `Function` `AllocateRegisters` method uses several `Function` methods such as `FindPointerVars` and `MakeRegistered` to find any variables which are accessed indirectly, and make a variable a `REGISTER` variable respectively.

After register allocation, the `Program` method `MatchBlockPatterns` is called. This method performs the control flow analysis of the designated resource pool. The information provided by this method is used as a guide by the GDL Emitter.

The final step in the analysis is the optimization and GDL emission. The optimization steps are built into the GDL Emitter. The constant shift operations are detected and eliminated inline in the GDL Emitter. Constant multiplication operations are detected inline by the GDL Emitter and are

## *CHAPTER 7. GDLGEN TOOL ORGANIZATION*

replaced by a `MultiTable` object which contains the supported constant multiplication operations. GDL Emission also occurs at this by using the `Program` method `EmitGDL`. This method is a shell which iterates over all of the `Function` objects in a resource pool and calls their respective `EmitGDL` method.

This outline is intended as a guide to program flow only. For more information please see the source code listing.

# Chapter 8

## Conclusions

This thesis has presented a tool, *gdlgen*, which analyzes and optimizes acyclic dataflow graphs for use in a hardware compiler. This tool takes a preliminary graphical description of a program and processes it to generate a GDL (Graph Description Language) description of the program. While processing the graphs, the tool analyzes the important features of the program including memory accesses and control flow information in order to generate a complete description of the program. Certain hardware specific optimizations are introduced to cause a more efficient hardware instantiation of the dataflow graphs making up a program. The output of this tool is given to the *SAS* tool which partitions and schedules the graphs across the multiple resource pools available on a CCM platform.

### 8.1 Successes of the tool

*Gdlgen* represents a major advance in the ability to compile C code to hardware. Previous efforts have limited the datatypes and control flow constructs which could be recognized by the compiler. The tool supports the use of `array` and `structure` datatypes which until now had not been supported. The breadth of support available from the combination of *gdlgen* and *lcc* allows nearly the entire C language to be supported.

## CHAPTER 8. CONCLUSIONS

### 8.2 Shortcomings of the tool

Although the tool handles most aspects of the input language, there are still some constructs which have not been implemented. A mechanism for instantiating balanced arithmetic DAGs from unbalanced DAGs would also enhance hardware parallelism. Also, although simple `switch-case` constructs are supported, there are still situations which will cause the tool to generate incorrect output. These situations include multiple `case` labels in `switch-case` constructs as well as `case` labels without a `break` statement. The tool should accommodate these revisions relatively easily.

Further, C `goto` statements are not supported, nor can they be since GDL does not provide a mechanism to perform unconditional jumps. `Continue` and some forms of `break` statements also cause this problem.

### 8.3 Future work

This is a very active area of research since CCM software solutions have yet to catch up with the advances in the hardware platforms. The future of Custom Computing Machines as a computing solution *depends* on the ability to provide application developers a standard, familiar interface with which to program them. In addition, this interface must be portable. The technology of CCM software solutions is moving ahead rapidly, but still has some distance to go in order to make CCM application development possible outside of the research environment. When this is possible, CCMs will truly have come of age.

## REFERENCES

- [1] A. Aho, *Compilers, Principles, Techniques and Tools*, Addison Wesley, Upper Reading, Massachusetts, 1986.
- [2] J. Armstrong, F. Gray, *Structured Logic Design with VHDL*, Prentice Hall, Upper Saddle River, New Jersey, 1993.
- [3] J. Arnold, D Buell, E. Davis, "Splash 2," Proceedings: 4th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 316-322, 1992.
- [4] P. Athanas, A. Abbott, "Real-Time Image Processing on a Custom Computing Platform," *IEEE Computer*, vol. 28, no. 2, pp. 16-24, Feb 1995.
- [5] P. Athanas, H. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis: Architecture and Compiler," *IEEE Computer*, vol. 26, no. 3, pp. 11-18, Mar 1993.
- [6] D. Buell, K. Pocek, "Custom Computing Machines: An Introduction," *The Journal of Supercomputing*, vol.9, no.3, pp. 219-29.
- [7] T. Drayer, W. King, J. Tront, R. Conners, "A Modular and Reprogrammable Real-time Processing Hardware, MORRPH," *IEEE Symposium on FPGAs for Custom Computing Machines* (P. Athanas and K. Pocek, eds.), IEEE, pp. 11-19, Apr 1995.
- [8] C. Fraser, D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, Redwood City, California, 1995.
- [9] B. Fross, "A Global Shared Memory Architecture for Splash-2 to Support HLL Compilers," Master's Thesis, Virginia Polytechnic Institute and State University, 1995.
- [10] D. Galloway, "The Transmogripher C Hardware Description Language and Compiler for FPGAs," *IEEE Symposium on FPGAs for Custom Computing Machines* (P. Athanas and K. Pocek, eds.), IEEE, pp. 136-144, Apr 1995.
- [11] S. Guccione, M. Gonzalez, "A Data-Parallel Model for Reconfigurable Architectures," *IEEE Workshop on FPGAs for Custom Computing Machines* (D. Buell and K. Pocek, eds.), IEEE, pp. 79-87, Apr 1995.
- [12] S. Gupta, "Predicting Execution Behavior for Codesign Synthesis Using Static Analysis," Master's Thesis, Virginia Polytechnic Institute and State University, 1995.

## REFERENCES

- [13] C. Iseli, E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis," *IEEE Symposium on FPGAs for Custom Computing Machines* (P. Athanas and K. Pocek, eds.), IEEE, pp. 173-179, Apr 1995.
- [14] W. Luk, "A Declarative Approach to Incremental Custom Computing," *IEEE Symposium on FPGAs for Custom Computing Machines* (P. Athanas and K. Pocek, eds.), IEEE, pp. 164-172, Apr 1995.
- [15] D. J. Magenheiner, L. Peters, K. W. Pettis, and D. Zuras, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 980-990, August 1988.
- [16] C. J. Neuhauser, "Instruction Stream Monitoring of the PDP-11," Stanford University Department of Electrical Engineering, Computer Systems Lab., Technical Note 156, May 1979.
- [17] M. Newman, W. Luk, I. Page, "Constraint-based Hierarchical Placement of Parallel Programs," Springer-Verlag, Proceedings, Field-Programmable Logic, Prague, Czech Republic, September 1994.
- [18] K. Paar, "A Custom Computing Machine Solution for Simulation of Discretized Domain Physical Systems," Master's Thesis in progress.
- [19] J. Peterson, R. O'Connor, P. Athanas, "Scheduling and Partitioning of ANSI-C Programs Onto Multi-FPGA CCM Architectures," *IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, Apr 1996.
- [20] P. Ross, "Moore's Second Law," *Forbes Magazine*, vol. 28, no. 2, pp. 98-99, Mar 1996.
- [21] B. Stroustrup, *The C++ Programming Language, 2nd Ed.*, Addison Wesley, Reading, Massachusetts, 1994.
- [22] A. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys*, vol. 18, no. 4, pp. 365-396, December 1986.
- [23] J. Wakerly, *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
- [24] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh, "PRISM-II Compiler and Architecture," *IEEE Workshop on FPGAs for Custom Computing Machines* (D. Buell and K. Pocek, eds.), IEEE, pp. 9-16, Apr 1995.
- [25] *The Programmable Gate Array Data Book*, Xilinx Inc., San Jose, California, 1994.
- [26] T. Halfhill, "Intel's P6," *Byte Magazine*, vol. 20, no. 4, pp. 42-58, Apr 1995.
- [27] D. Pountain, T. Halfhill, "CPU Scorecards," *Byte Magazine*, vol. 20, no. 11, Nov 1995.

## *REFERENCES*

- [28] B. Smith, "Ultrafast UltraSparcs," Byte Magazine, vol. 21, no. 1, pp. 139-140, Jan 1996.
- [29] Annapolis Micro Systems, 190 Admiral Cochrane Dr. Suite 130, Annapolis, MD 21401.
- [30] Giga Operations, 2374 Eunice St., Berkeley, CA, 94708.
- [31] Virtual Computer Corporation, 6925 Canby Av. #103, Reseda, CA, 91355.

## VITA

**R. Brendan O'Connor** was born in Baltimore, Maryland on March 24, 1972. He earned the Bachelor of Science Degree in Computer Engineering with a Minor in Computer Science from Virginia Tech in May, 1994. He joined the graduate program at Virginia Tech in August, 1994 to pursue his Master's Degree in Electrical Engineering. His interests include VLSI design and Design Automation tools, as well as some occasional golf and snowboarding. Brendan is going to be joining Intel Corporation in Portland, Oregon as a Component Design Engineer in July, 1996.