DISSERTATION

COMPILING DATAFLOW GRAPHS INTO HARDWARE

Submitted by

Robert E. Rinker

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2005

COLORADO STATE UNIVERSITY

October, 2005

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY ROBERT E. RINKER ENTITLED COMPILING DATAFLOW GRAPHS INTO HARDWARE BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Adviser

Department Head

ABSTRACT OF DISSERTATION

COMPILING DATAFLOW GRAPHS INTO HARDWARE

Conventional computers are programmed by supplying a sequence of instructions that perform the desired task. A reconfigurable processor is "programmed" by specifying the interconnections between hardware components, thereby creating a "hardwired" system to do the particular task. For some applications such as image processing, reconfigurable processors can produce dramatic execution speedups. However, programming a reconfigurable processor is essentially a hardware design discipline, making programming difficult for application programmers who are only familiar with software design techniques.

To bridge this gap, a programming language, called SA-C (*Single Assignment C*, pronounced "sassy"), has been designed for programming reconfigurable processors. The process involves two main steps - first, the SA-C compiler analyzes the input source code and produces a hardware-independent intermediate representation of the program, called a *dataflow graph* (DFG). Secondly, this DFG is combined with hardware-specific information to create the final configuration.

This dissertation describes the design and implementation of a system that performs the DFG to hardware translation. The DFG is broken up into three sections: the data generators, the inner loop body, and the data collectors. The second of these, the inner loop body, is used to create a computational structure that is unique for each program. The other two sections are implemented by using prebuilt modules, parameterized for the particular problem. Finally, a "glue module" is created to connect the various pieces into a complete interconnection specification.

The dissertation also explores optimizations that can be applied while processing the

DFG, to improve performance. A technique for pipelining the inner loop body is described that uses an estimation tool for the propagation delay of the nodes within the dataflow graph. A scheme is also described that identifies subgraphs with the dataflow graph that can be replaced with lookup tables. The lookup tables provide a faster implementation than random logic in some instances.

Robert E. Rinker
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Fall 2005

ACKNOWLEDGMENTS

I would like to first thank my advisor, Dr. Walid Najjar, and the members of my committee, Dr. Wim Böhm, Dr. Dale Grit, and Dr. Anura Jayasumana, for putting up with me over all these years. You have all provided me with invaluable counsel throughout my term as a PhD student.

Next, thanks to all my fellow students who worked on the Cameron project, for providing such an amazing work environment. To Jeff Hammes - I will always remember and relish our cubicle discussions. To Margaret Carter, who implemented an amazing DFG to VHDL translator. To Amit Patel, who did some incredible work on the VHDL reduction operator library. To Monica Chawathe, who was always there, ready to fix bugs in the translator code. To Charlie Ross, whose incredible coding expertise I will always admire. To Harish Kantemneni and Dave McClure, who put together and maintained the DFG viewer. And to Danny Kulkarni, who worked on the DFG space estimator. It was all of you who made this project so enjoyable!

Finally, I would like to thank my colleagues at the University of Idaho for their support while I was writing my dissertation, and especially Dr. John Munson and Dr. Bob Hiromoto, for struggling through some of the early drafts, and providing invaluable suggestions on how to improve them.

DEDICATION

I dedicate this dissertation to my family, including my parents, Ed and Phyllis Rinker, and my children, Jeff, Gina, and Jodi, for providing the moral support that I needed to follow through with this project. You will never know how much that support means to me. Thanks!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A conventional processor operates by executing a sequence of instructions (a "program") on a fixed configuration of hardware - the "datapath." Each instruction generates a set of signals that controls the function of this fixed datapath; since the datapath is fixed, each instruction must specify a similar sequence of steps, usually in synchronization with a clock that determines a common execution speed for each instruction. By contrast, a Reconfigurable Computer System (RCS) consists of a set of hardware resources that can be interconnected to solve a specific calculation. In an RCS, the "program" consists of this interconnection specification. The advantage of this approach is that the resulting configuration can be optimized for the particular problem - the speed is determined by the propagation delay of the resulting circuit, rather than a fixed clock. As a result RCS's promise significant performance gains over conventional processors, especially for highly parallel applications like image processing.

However, current design environments for RCS's require a considerably different skill set than the typical application programmer possesses. Instead of a programming language such as C, or even assembly language, RCS's usually require an intimate knowledge of the hardware characteristics, including specific timing information. In a coprocessor-based system, both software knowledge for the host–based part of the application and hardware knowledge for the reconfigurable computing system. The lack of this dual knowledge generally requires that the design be partitioned quite early in the design process into software and hardware parts, and the two parts handed off to two different skill groups. This early partitioning requires that a significant amount of effort be placed into the

interface between the two, even before all the tradeoffs are known.

The Cameron project (Najjar, Draper, Böhm, and Beveridge 1998, `www.cs.colostate.edu/cameron`) is a project whose goal is to integrate these two parts into a single design system, and to allow the design to be carried out entirely by the application programmer, without requiring intimate knowledge of hardware, timing or interface details. The project involves the design of a language which is particularly suited for translation into hardware, called SA-C (*Single Assignment C*, pronounced *"sassy"*). This dissertation describes a portion of the work done in the design and implementation of this language.

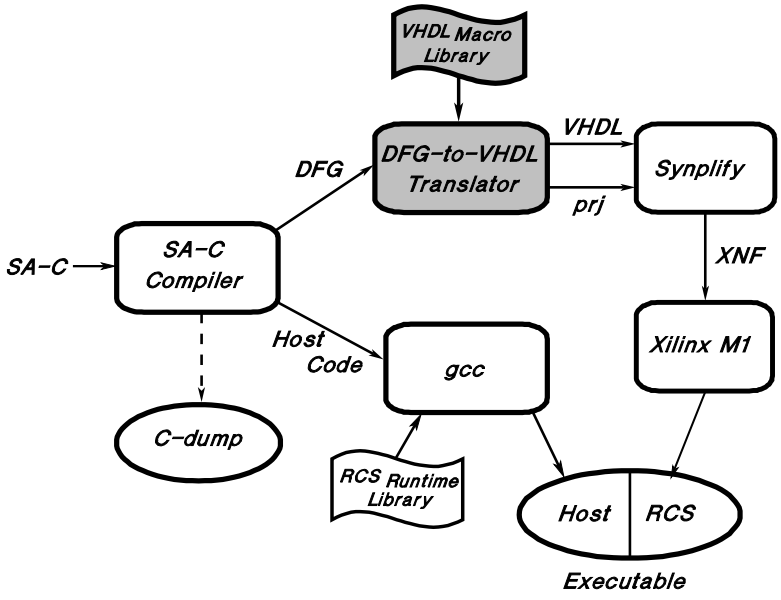


Figure 1.1: Overview of the SA-C Compilation process

1.1 The SA-C Compiler

An overview of SA-C the compilation system is shown in figure 1.1. The compiler accepts SA-C source programs and compiles them, producing two main outputs. The first is the host code - a program that runs on the host workstation; its main purpose is to initialize and control the operation of the attached RCS. The second output is an intermediate form called a Dataflow Graph (DFG). The DFG is a graphical representation of that part of the SA-C program that executes on the RCS. It shows the relationship between the processing

elements required for the calculation in a hardware independent form, and without any timing information.

The SA-C compiler performs numerous aggressive optimizations, some of which are typical compiler optimizations and some which are unique to its role as a compiler designed to target RCS systems, before it creates the DFG. Some more information about the compiler is provided later; a more detailed description of the compiler and its implementation is provided in Hammes 2000.

To produce a complete RCS program, the DFG is further processed by a DFG-to-VHDL translator. VHDL (**V**HSIC **H**ardware **D**escription **L**anguage, where VHSIC stands for **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) is a standard hardware specification language that is used as the starting point by chip designers to describe their hardware designs. Its use requires a thorough understanding of the hardware that it is describing - application programmers typically do not possess these skills. The DFG-to-VHDL translator incorporates these details into the VHDL, so the programmer can create an application without having to be concerned with them.

The resulting VHDL code is synthesized and place-and-routed, using commercial software tools, to create the binary executable for the RCS. A final RCS program, therefore, consists of the host executable and the binary code that programs the RCS.

1.2 Purpose of this Dissertation

While the SA-C compiler is aggressive in its optimizations, it works with no specific knowledge of the hardware it is targeting. The DFG it produces is intentionally designed to be hardware independent; it is analogous to the "front end" (i.e., the lexical scanner and semantic analyzer) of a traditional compilation system. The DFG serves as the intermediary between this front end and the "back end," or code generator. This dissertation focuses on the back end. In particular, it focuses on the implementation and optimizations of the DFG-to-VHDL translator; this portion of the process is highlighted in figure 1.1. Again, comparing the process with a traditional compiler, where the back end produces assembly language that can be converted to an executable using the system's assembler and

linker/loader, here the "assembly language" is VHDL, which is converted to an executable using commercial synthesis tools.

The first requirement for any compiler, and for the DFG-to-VHDL translator in particular for this project, is that it produce correct configurations from the DFG. Thus, this dissertation first concerns itself with that task.

In addition to the hardware independent optimizations performed by the SA-C compiler itself, there are other hardware dependent optimizations that can be applied at the DFG level to improve the performance of the final design. Pipelining allows a DFG that has a long propagation delay to be executed in segments, thereby reducing the delay through each segment and allowing an increase in the overall clock speed. Unlike a conventional processor, where the execution pipeline is fixed, in an RCS the pipeline can be created as part of the hardware configuration. It is the responsibility of the compiler to determine the best way to implement the pipeline.

Lookup tables have long been used to implement complicated calculations for high speed and with a minimum of hardware. In fact, lookup tables are used at the very lowest level in Field Programmable Gate Arrays (FPGAs), a technology that is frequently used in RCS's. It may be possible to improve the performance of a design by substituting lookup tables for DFG segments.

The remainder of this dissertation is organized as follows. Chapter 2 presents an overview of the field of reconfigurable computing, including some background, and a short survey of pertinent hardware and software research in the field. The third chapter describes a manual (not automated) implementation of a popular image processing algorithm - the Prewitt edge detecting algorithm - on the Annapolis Microsystems Wildforce board. This manual implementation helped in better understanding the behavior of the hardware, the software design tools, and allowed some experimentation with different design methods that would later become important when the process was automated. The manual implementation also provides a performance target for the automated system to achieve. Chapter 4 describes the design and implementation of the first automated compilation system, which translates the DFG representation into VHDL. Chapter 5 describes the

work done to determine propagation delay estimates for each DFG node, and the results of using these estimates to pipeline the inner loop body. Chapter 6 describes research done in replacing appropriate sections of the inner loop body with a lookup table, in order to increase execution performance. Finally, chapter 7 concludes and suggests future work.

When incorporated into the rest of the SA-C compilation system, the work described in this dissertation allows an application programmer to transform a SA-C source program into an executable for an RCS in a single operation. Using the compiler requires no particular hardware knowledge of the underlying RCS. However, optimizations provided by the compiler front-end, as well as the optimizations described here, allow the programmer considerable freedom to experiment with the final design to improve performance. Using the SA-C compiler to produce a design, including optimizations, only requires a few days of work, compared with the several months that are required using manual methods.

Chapter 2

Background

The idea of configurable hardware is not new. Estrin (Estrin 1963) describes a *Variable Structure Computer System* in 1963; an IBM 7090 computer (the "fixed structure" computer) was outfitted with an array of plug boards on which could be placed flip-flops and diode-based logic functions. He recognized the parallelism that existed in certain computations, and predicted a speedup of from 2.5 to 1000 in certain numerical operations when using the proper configurable system. His idea was that the configurable part of the system would come wired with a "standard state" which would compute things like square root and trigonometric functions; once the computational bottlenecks in a program were discovered, this standard state could be modified to more specifically compute the results for the problem being solved. The idea of using plug boards for the variable structure computer came from the "wire programs" that many computers of the time were still using to perform their tasks; the "stored program" concept was still relatively novel at the time. Many of the concepts proposed by Estrin's work were far ahead of its time; many are resurfacing now because technology exists that can support them.

Modern reconfigurable computing came about with the advent of the Field Programmable Gate Array (FPGA) in 1984 (Xilinx, Inc., San Jose, CA. 1998). This technology marries the electrically programmable capability of digital logic chips which implement simple AND-OR logic functions, such as PROMS (Programmable Read Only Memories, which contain a fixed AND array and a programmable OR array) and PALs (Programmable Array Logic, with a programmable AND array and fixed OR array), with the more complex gate array, which consist of hundreds or thousands of gates that can be

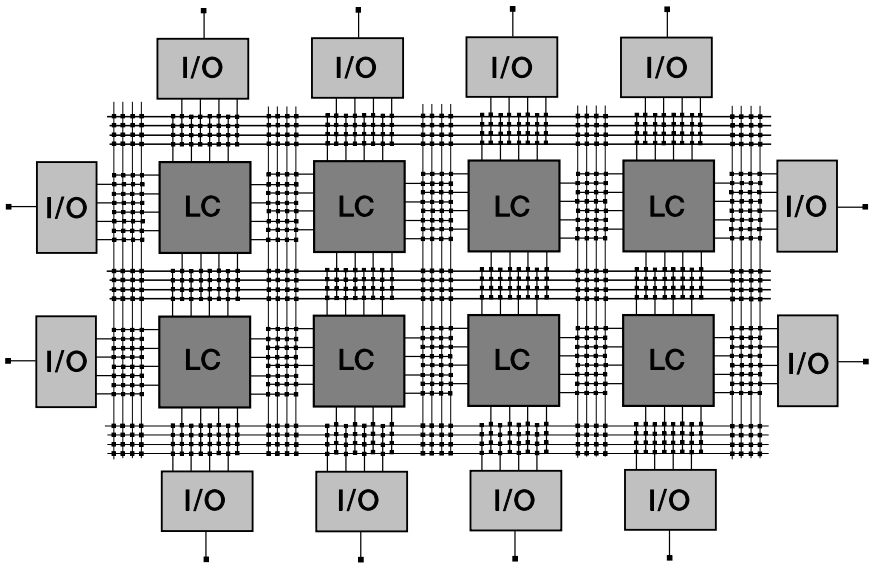


Figure 2.1: Architecture of an FPGA. In the picture, `LC` stands for logic cell, `I/O` stands for an I/O block, and the small black squares represent programmable connection points.

interconnected in any configuration by designing a custom interconnection mask.

FPGAs are the basis for many of the reconfigurable computing engines in use or being planned today; other designs incorporate FPGA-like features along with other integrated circuit technology, such as RISC or special-purpose processors. The rest of this chapter provides a survey of current reconfigurable research. In Section 2.1, the basic structure of the FPGA and its evolution are discussed. Section 2.2 reviews several hardware implementations which use commercially-available FPGAs. A couple of custom reconfigurable computing projects are described in Section 2.3. Finally, Section 2.4 discusses some software-oriented reconfigurable computing projects.

2.1 FPGA Technology and Evolution

Figure 2.1 shows the basic architecture of an FPGA. There are three types of circuit components inside the FPGA:

- Logic blocks, laid out in a two-dimensional array, that can be programmed to perform a certain logic function.

- I/O blocks, around the outside of the chip, that provide the interface between the

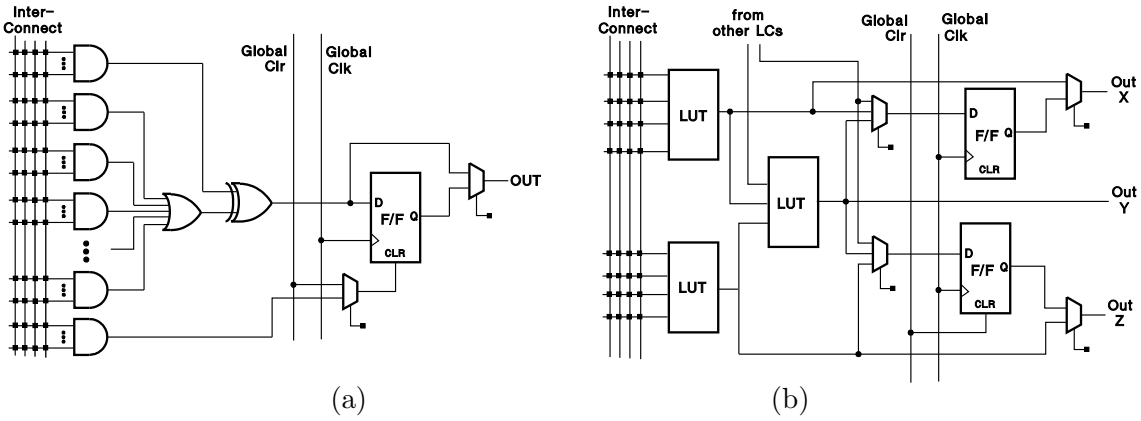


Figure 2.2: Two types of logic cells used in FPGAs. (a) Gate-based logic cell. (b) Lookup table-based logic cell.

interior and exterior of the chip.

- An interconnection grid, that can be used to connect the logic blocks to each other.

Each logic block consists of a small amount of digital logic; it is the organization of this logic block that distinguishes one manufacturer's chip, or even one family of chips from the same vendor, from another. The makeup of this logic block falls into one of two types. The first type consists of a simple fixed logic function (usually some variation of an AND-OR array) whose specific logic function – the exact interconnection within the function – can be programmed. This type is common in lower capacity chips – it has proven to be a significant technological challenge to use this type of logic block for more dense designs. The second type of logic block uses a lookup table, where a Boolean function of several variables can be specified. The logic block also contains one or more Flip Flops, so the value generated by the logic can be stored. This type has proven to be more scalable, due to the inherent circuit regularity of the lookup table. An example of the two main types of logic blocks are pictured in Figure 2.2.

I/O blocks can be programmed to be inputs, outputs or both. They provide the interface to external circuits, and can be configured to provide different electrical characteristics to the external circuit (pull-up resistors, rise time limits). Finally, they can provide latches between the internal and external connections.

The interconnection network between logic and I/O blocks is accomplished by specifying the state (either connected or unconnected) of tie points on the interconnection grid. Usually several types of interconnect between blocks are available – short lines for connecting local logic blocks, long lines for connecting logic blocks that are far apart on the chip, and global (buffered) lines that attempt to minimize propagation delays across the chip (usually used for clock signals). Manufacturers try to provide a versatile interconnect, since the routing of signals within a chip is a very time-consuming process, often taking hours even when automated.

The first FPGAs were programmed via *antifuses*, which are small fusible links that are initially placed everywhere a possible connection may be needed. The programming consists of an electrical procedure that literally "blows up" those connections that are not desired, leaving only the desired connections. These FPGAs can only be programmed once, since the fuses that have been disconnected cannot be restored. As such, this type of FPGA is not well suited for reconfigurable purposes. Due to the destructive nature of the programming, fuse-programmable FPGAs are usually limited in circuit density - literally, space must be placed around the fuse to prevent unintended circuit destruction around the fuse.

The second method of programming is to use static RAM (SRAM) cells for the programming. In this situation, the SRAM cell is set to 0 or 1, depending upon whether the connection is to be made or not. The SRAM cell is more compatible with surrounding circuitry, so much higher densities can be achieved. The FPGA is programmed by supplying a string of bits – one for each SRAM cell – to the FPGA while it is in a special "program" mode. This string is supplied either by an external ROM chip, or in the case of a reconfigurable system, by a host computer. Since the programming process is non-destructive, the chip can be reprogrammed any number of times simply by changing the sequence of bits that is downloaded.

In the last few years, the evolution in FPGAs has followed much the same path as microprocessors, partly because the same technology benefits each type of chip. Within the same family of FPGA chips, the size of the logic block matrix has increased. New

families of FPGAs have also been developed, offering an order of magnitude increase in both size and speed over previous family. For example, the Xilinx Virtex (Xilinx, Inc. 1999) is the successor to the Xilinx XC4000 (Xilinx, Inc., San Jose, CA. 1998) series. The high end of this new series contains 10 times the number of configurable logic blocks (CLBs, Xilinx's name for its logic blocks) and can operate at 4 times the speed (200 vs 50 MHz) as the XC40150, the high end of the previous generation. This increase in performance has a profound impact on the weighting that needs to be applied to the various criteria within a given design.

One unfortunate aspect of FPGAs, insofar as reconfigurable computing is concerned, is that their primary application is **not** reconfigurable computing. The primary market for FPGAs is to serve as "glue logic" in circuit designs as prototype "breadboards" to test circuit designs before they are committed to a more permanent form such as a custom chip or Application Specific Integrated Circuit (ASIC). They are not optimized for use in reconfigurable applications. This fact is particularly evident during reprogramming. The usual method of programming is to supply the programming bits, in bit serial fashion, to the FPGA. Only one pin is used for this programming, thereby providing more pins for I/O between the FPGA and external circuits. Also, for the most part the entire chip must be reconfigured – partial reconfigurability is not usually possible, even if only a portion of the chip is being used or needs to be reconfigured. The total reprogramming time is in the order of milliseconds; this can be a dominating amount of time when considering the entire amount of time required for complete execution of an application. One series of chips that was designed specifically for reconfigurable computing, the XC6200, allowed the configuration array to be accessed like normal random access memory, 16 bits at a time. However, it has not been successful in the marketplace. This circumstance is reflected in some of the compromises that appear in reconfigurable computing designs using these commercial chips.

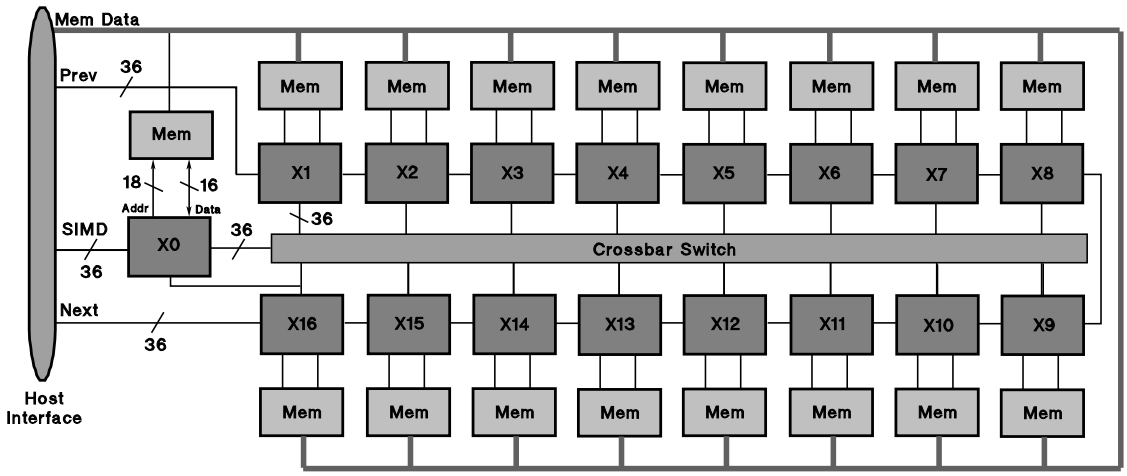


Figure 2.3: Simplified Architecture of the Splash-2 Configurable logic board

2.2 Survey of Reconfigurable Computing Hardware which Use Off-the-Shelf FPGA Chips

In spite of the limitations, several reconfigurable computing boards have been designed using commercially available FPGA chips (Sanchez, Sipper, Haenni, Beuchat, Stauffer, and Perez-Uribe 1999); this class of boards is sometimes referred to as *netlist architecture* (Mangione-Smith 1997b), since a primary intermediate form for specifying the programming of such boards is an interconnection specification called a netlist. The sub-sections that follow discuss several of these systems; the designs span more than a 10 year period (roughly from 1990 through 2002), and represent a considerable evolution in design. All of the designs represent a coprocessor paradigm - the reconfigurable board is attached via a high speed bus to a standard processor, which acts as the control and management unit for the coprocessor board. They are of particular interest here because this type of hardware is the target for the work described in this dissertation.

2.2.1 The Splash-2 Reconfigurable Computing Board

One of the first reconfigurable processing boards, at least in the modern era, The Splash-2 (Buell, Arnold, and Kleinfelder 1996) was developed in 1991, and has been used in several different applications. A simplified diagram of the Splash-2 is shown in Figure 2.3.

The Splash-2 utilizes 16 Xilinx XC4010 FPGAs (Processing Elements, or PEs, labeled

X1 through X16), connected to each other via a 36 bit bus (intended to be used as 32 bits of data, plus 4 user-definable tag bits) in a linear array. Each PE is also connected to its own 512K byte local memory, organized as 16 bit words. One additional FPGA (X0) serves as the control processor – its main job is to control the *crossbar*, a programmable, bidirectional 36 bit interconnect which is connected to the other processing elements. This crossbar can be reconfigured as often as every clock cycle, if desired, by specifying one of eight pre-determined configurations. Each configuration allows a PE to broadcast data to one or more of the others; the data paths can be configured in four-bit chunks.

The system is attached to a Sun SPARCstation host processor, via the processor's S-Bus. The host is responsible for downloading the FPGA configurations, setting the board frequency and controlling the clock, and passing data to and from the ends of the linear array. The host can also load data into each local PE memory. The role of the host can either be synchronous, directly controlling the Splash-2 while it is operating, or asynchronous, allowing the Splash-2 to function independent from the host, once initialized.

The system can be expanded by adding additional boards. The additional PEs are connected so they add to the length of the linear array.

A variety of applications have been implemented on the Splash-2, including searching (Hoang 1993; Pryor, Thistle, and Shirazi 1993), pattern matching (Ratha, Jain, and Rover 1996), convolution (Ratha, Jain, and Rover 1995) and image processing (Athanas and Abbott 1994). The speedups of these applications compared with similar applications executing on conventional processors can be impressive, with a range of from less than 1 (no speedup) to over 1000, relative to an algorithm executing on a conventional processor. Not surprisingly, the speedup in general depends upon how closely the application matches the processing strengths of the Splash-2. Since the Splash-2 consists of 16 fairly small-size FPGAs (the XC4010 contains a matrix of 20×20, or 400 total, CLBS, for a total of 6800 CLBs on each board), those applications which can be partitioned into up to 16-deep pipeline stages, with each stage doing a simple operation that can fit into one of the FPGA chips, achieve good speedups over sequential processors. Similarly, applications which perform small bit-width operations, which exploit the fine-grain nature of FPGAs,

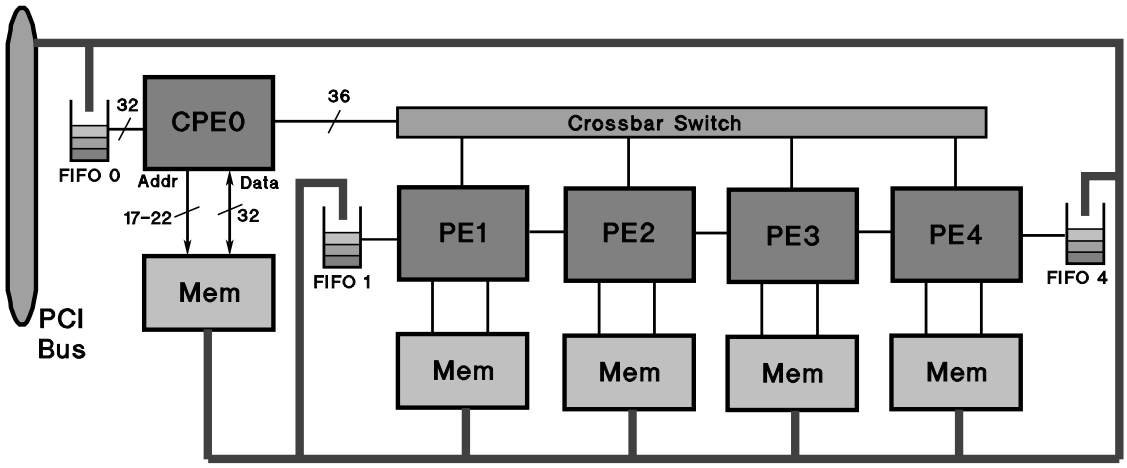


Figure 2.4: Simplified Architecture of the Wildforce Configurable logic board

and would be inefficient on a byte-at-a-time processor, do very well. For example, the largest speedup was reported on a bit matching application that also used the PEs on Splash very effectively as a systolic array. Other applications which involve complicated operations, or more sequential processing steps, do not perform as well. Some of the compromises made during the design of the Splash-2, such as a narrow 16 bit memory data bus (a limitation imposed by a lack of I/O pins on the FPGAs), limit its effectiveness in some applications.

The Splash-2 was one of the first implementations of a reconfigurable computing board. It showed impressive performance in a wide range of applications, and sparked considerable interest in reconfigurable computing. Even more importantly, it has provided a platform to experiment with reconfigurable computing architectures. Many of the "good" features of the Splash-2 have been carried over in new designs, and some of its limitations have been eliminated or otherwise addressed in newer architectures.

2.2.2 The Wildforce Reconfigurable Computing Board

The Annapolis Micro Systems Wildforce^{TM} Computing Board (Annapolis Micro Systems, Inc., Annapolis, MD 1997) draws considerable inspiration from the Splash-2. It is one of a family of commercially-available boards which also utilize off-the-shelf components. It was introduced in 1997. A simplified diagram of its architecture is shown in Figure 2.4.

The Wildforce processing board includes a control processor (called CPE0) which controls a crossbar with similar capabilities to that of the Splash-2. The main processing on the board is again done with a linear array of FPGAs (PE1 - PE4), each with its own local memory. The primary data paths, this time including the memory data bus, are all 36 bits each.

However, there are also significant differences with the Splash-2. The control processor and PEs are implemented with later and much larger members of the Xilinx XC4000 family, with a choice from the XC4025 (a 32×32 matrix, 1024 CLBs) up to the XC4085 (56, or 3136 CLBs) for each PE. Because of the increased logic density, a smaller number of chips (4 vs 16) can be used for the linear array, but the total number of CLBs available is on par, and in some cases, much larger than, the total number on the Splash-2. (For example, the mid-size Wildforce board owned by Colorado State University contains five XC4036 chips, which contain a 36×36 CLB matrix, for a total of 1296 CLBs on each chip, and 6480 total CLBS, comparable to the 6800 CLBs on the Splash-2.) The higher density chips reduce the number of interfaces required in the linear array, and allow the Wildforce to be used for shallower, but more complex, pipeline designs. Yet, nothing precludes the placement of several pipeline stages on each FPGA, thereby allowing Splash-2 applications to be mapped to the Wildforce.

New features facilitate the movement of data between the host and the Wildforce. The PCI bus interface allows DMA mode, which supports bandwidths up to 132 MB/Sec between host memory and the board. Three hardware FIFOs, each 512 words deep, are also implemented in hardware; they facilitate the streaming of data to and from the host. Finally, an external I/O interface can be connected to the board, allowing data to be sent to/from the board without any host intervention.

Since this board is the target of some of the work described in this dissertation, a few observations about its strengths and limitations, along with performance data on several applications developed during the work described in this dissertation, will be presented later.

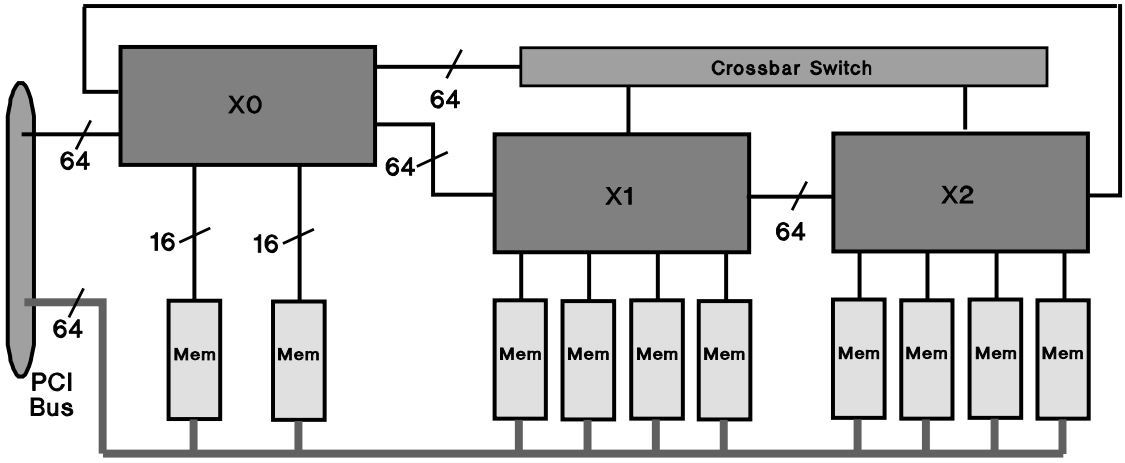


Figure 2.5: Architecture of the SLAAC Configurable Logic Board

2.2.3 The SLAAC-1 Reconfigurable Computing Board

The SLAAC-1 processor continues the evolutionary path from the Splash-2 to new technology. SLAAC (Systems Level Applications of Adaptive Computing) project (Schott, Crago, C., Czarnaski, French, Hom, Tho, and Valenti) is a comprehensive research project which defines its mission to be:

1. To define an open, distributed, scalable, adaptive computing systems architecture.

2. To design, develop, and evolve scalable reference platform implementations of this architecture, and

3. To validate the approach by deploying technology in multiple defense application domains.

As part of the project, the SLAAC-1 reconfigurable computing board has been developed. A diagram of the SLAAC-1 board is shown in Figure 2.5.

The SLAAC-1 represents another step in the evolution of the architecture initiated by the Splash-2. It is composed of 3 XC40150 chips, each of which containing a matrix of 72×72 (5184 total) CLBS, for a system total of 15552 CLBS – roughly 2.5 times that of the Splash-2. A second generation of the board is also being developed using the newer Virtex family of FPGAs (the XCV10000); this will increase the total CLB count to over 18,000, and should allow up to a four-fold increase in processing speed. The concept of one

FPGA (X0) serving as the control processor is carried over from the Splash-2, as does the crossbar between the FPGAs. The linear array connections continue to exist, although now the control processor is connected to the linear array from both ends, forming a ring among the three PEs.

All of the datapaths on the SLAAC-1 have been increased to 64 bits, with the crossbar and linear array buses expanded to 72 bits (64 bits plus 8 tag bits). The memory access bottleneck that exists in both the Splash-2 and Wildforce has been mitigated by allowing the memories to be treated either as a single 64 bit wide memory (32 bits for X0), or as four separate 16 bit memories (two for X0). The memories allow reads to occur in one cycle, versus two cycles for Splash-2 and Wildforce.

The SLAAC-1 continues the evolutionary path from the Splash-2 to new technology. It appears that with faster chips, 64 bit datapaths, and a more versatile memory structure, it will provide performance several times that of previous reconfigurable systems. It also continues the trend toward the use of fewer, denser chips, which reduces the number of external interfaces the programmer must contend with.

2.2.4 The Annapolis Wildstar Reconfigurable Computing Family

The next generation of reconfigurable computing boards developed by Annapolis Micro Systems is called Wildstar^{TM}. The name refers to a family of boards, as well as one of the members of the family. The processors consist of one or more Xilinx Virtex FPGAs. This chip provides up to a two order of magnitude increase in the number of logic elements (CLBs) when compared with the Xilinx XC4000 series chips, as well as a factor of four or more improvement in clock speed over the previous generation Xilinx XC4000 series chips. Continuing the evolutionary trend in FPGA-based computing boards mentioned earlier, this allows a dramatic improvement in the size of problem that can be computed on the board, yet reduces the number of external interconnections between chips, thereby reducing the task of partitioning a problem amongst several separate chips.

The block diagram of the largest member of the Wildstar family, also called the Wildstar (Annapolis Micro Systems, Inc., Annapolis, MD 1999b), is shown in Figure 2.6. It

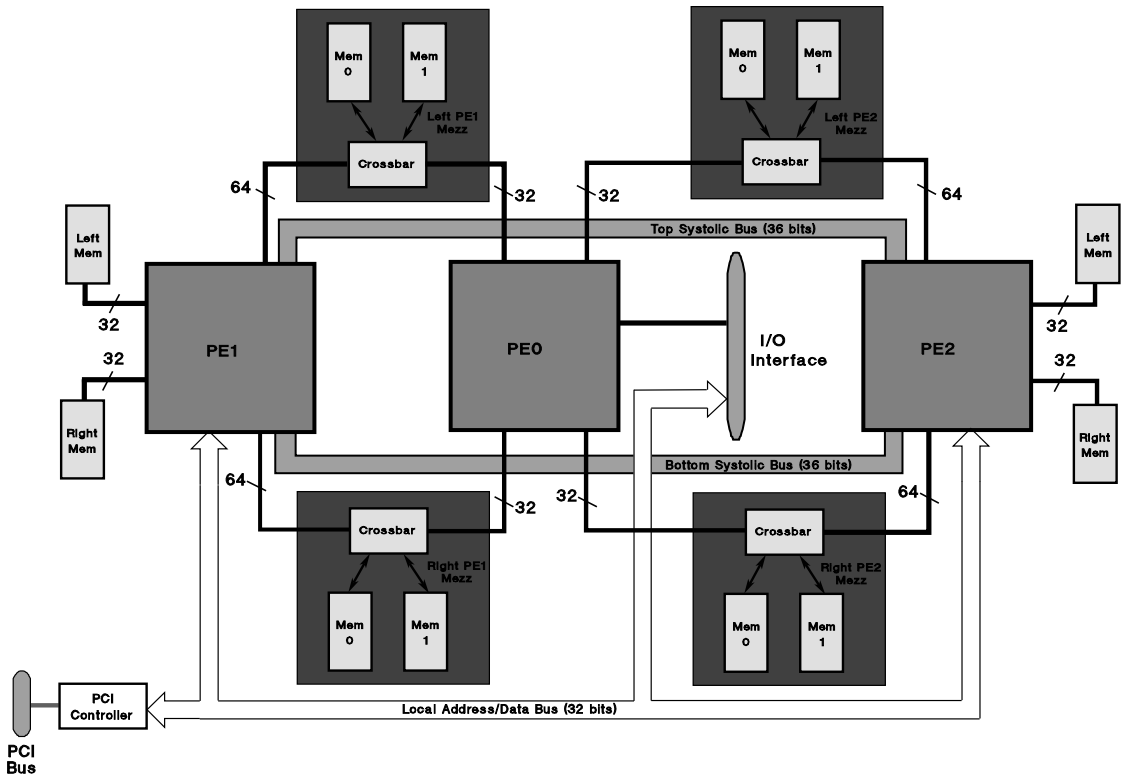


Figure 2.6: Block diagram of the Annapolis Wildstar Reconfigurable Computing board

consists of three Virtex FPGAs, each containing logic equivalent to 1 million gates. The architecture is symmetric around elements PE1 and PE2, while PE0 is primarily used to communicate with a possible input/output board that can be connected to the I/O interface. PE1 and PE2 each connect to two local memories, where data that doesn't need to be shared with the other processing elements can be stored. Additional memories, expandable via the use of mezzanine cards, sit between the processing elements; these memories can hold data that either needs to be accessible to more than one PE at a time, or that needs to be communicated between two PEs. A crossbar switch controls which PE has access to a given memory at a given time. Data can also be communicated between PE1 and PE2 via two 36 bit systolic busses.

A host processor can communicate both commands and data to the board via its PCI bus, which connects to the Local Address and Data (LAD) Bus within the Wildstar; this bus is connected to PE1 and PE2. Because of the many separate data paths between the processing elements and the various memories, a very impressive aggregate data bandwidth

between PEs and memory of 6.4GHz is possible.

A couple of smaller boards are also part of the Wildstar family, the Starfire board (Annapolis Micro Systems, Inc., Annapolis, MD 1999a) consists of a single Virtex Processing element - essentially PE1 in Figure 2.6 above. This board only contains a third of the number of CLBs as the Wildstar board. Nonetheless, when outfitted with a Virtex XCV10000 chip it provides about 30 times the number of equivalent gates as the entire earlier generation Wildforce board. A Starfire board was available during the latter portions of the research described in this dissertation, and it was used to run some of the larger optimization experiments. More specific information concerning the architecture of this board is discussed later in conjunction with these experiments.

Finally, a third member of the family, called Starfire-II, contains two processing elements - essentially PE0 and PE1 in Figure 2.6. This model is useful if a significant amount of I/O processing, performed by PE0, needs to be performed, but the full size of the Wildstar board is not required.

2.3 Custom-Designed Reconfigurable Computing Boards

While the technology used in commercially-available FPGAs has seen considerable improvement over the course of the last decade, and provides a much quicker and cheaper solution in the design of reconfigurable computing boards, nonetheless it offers some severe compromises in the design. There are two major problems with the off-the-shelf systems:

- They must be connected to the host processor via an external bus (such as VME or PCI), which limits data transfers between host and board to at most a few hundred megabytes/sec. Reconfigurable computing could benefit from bandwidths similar to that currently available between host and main main memory, perhaps in the gigabyte range.

- The currently available FPGA chips are not primarily designed for reconfigurable computing, and therefore are not optimized for reconfigurable applications. In particular, it requires milliseconds to download new configurations, and in general the

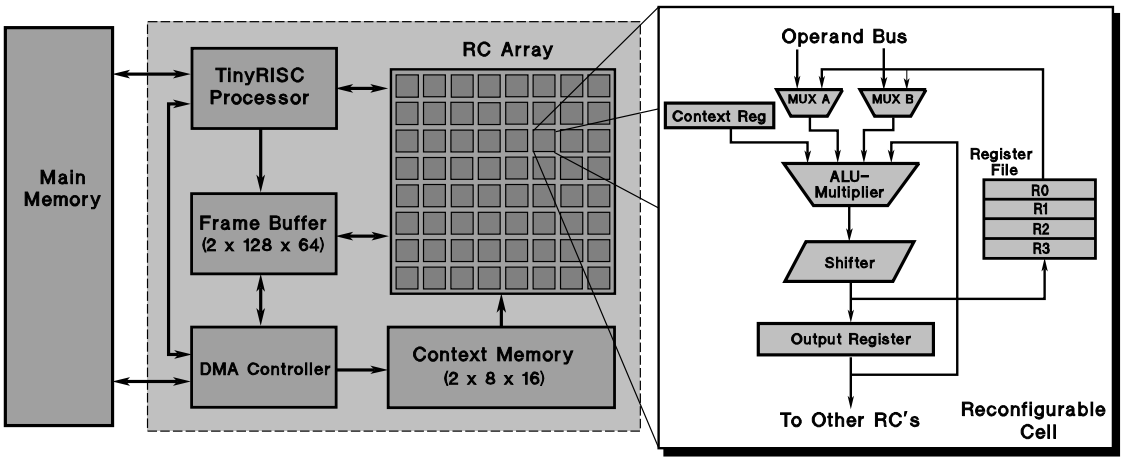


Figure 2.7: Architecture of the Morphosys Architecture. One of the Reconfigurable Cells (RC) within the RC Array is shown.

entire chip must be reconfigured, rather than only small portions as might be useful in reconfigurable applications. The ability to store multiple contexts in the chip at the same time is not possible.

Several projects are ongoing to develop custom reconfigurable systems A small survey of some of these systems, chosen to be representative of current research and development in the reconfigurable computing area, follows. Other projects are described in the literature (Hauser and Wawrzynek 1997).

2.3.1 Morphosys

The Morphosys project (Lu, Singh, Lee, Bagherzadeh, and Kurhadi 1999) is creating a new chip design that provides a CPU coupled with a specialized reconfigurable matrix of *reconfigurable cells* (the RC array). The overall architecture is shown in Figure 2.7. At first glance, the configuration looks similar to that of the attached processors discussed earlier; however, in this case, the processor is considerably more tightly coupled with the RC array, in that it can change the configuration context of the RC array as often as every clock cycle. Also, the results from the RC array are available immediately in the *frame buffer*, which is an integral component along the processor's data path.

In contrast to the fine-grain capabilities of the simple logic cells available in an FPGA, the RC array within Morphosys consists of a matrix of considerably coarser grained units

called Reconfigurable Cells (RC). A diagram of an RC is also shown in Figure 2.7. It consists of a preconnected set of components, including an ALU-Multiplier, a shift unit, input MUXs, a register file, and a context register. In short, each RC contains the essential component of a small processor. This type of reconfigurable hardware is referred to as a *chunky* architecture (Mangione-Smith 1997b), and stands in contrast to the more "fine grain" characteristics of the *netlist* architectures represented by conventional FPGAs. Instead of connecting gates together to implement low-level logic functions, the programming of the RC array within Morphosys is more analogous to connecting processor functional units together to perform custom operations. Programming should be relatively quick, since it is being done at a much higher level than traditional FPGAs. The RC array consists of an 8×8 matrix, a total of 64 components instead of the thousands of logic cells in a conventional FPGA.

The first implementation of the Morphosys architecture is called the M1. The second round of implementation, the M2 chip, is currently running in simulation at 450MHz.

2.3.2 The RAW Project

The RAW Project (Waingold, Taylor, Srikrishna, Sarkar, Lee, Lee, Kim, Frank, Finch, Barua, Babb, Amarasinghe, and Agarwal 1997) is a comprehensive project which involves the development of both hardware and software for reconfigurable computing; the hardware component will be highlighted here.

The RAW architecture is pictured in Figure 2.8. It marries a CPU and configurable logic together in a single unit called a *tile*, in an attempt to minimize the distance between the host and reconfigurable logic that plagues the attached processors. The tiles are then replicated into a matrix on a single chip, with an interconnection network between them.

Each tile consists of several components: an instruction memory (IMEM), similar to the instructions executed by a conventional processor, data memory (DMEM), a fairly conventional ALU, the configurable logic (CL), and a programmable switch along with its associated program memory (SMEM). The placement of the configurable logic into the tile's primary datapath allows data to be transformed in custom ways as it is being

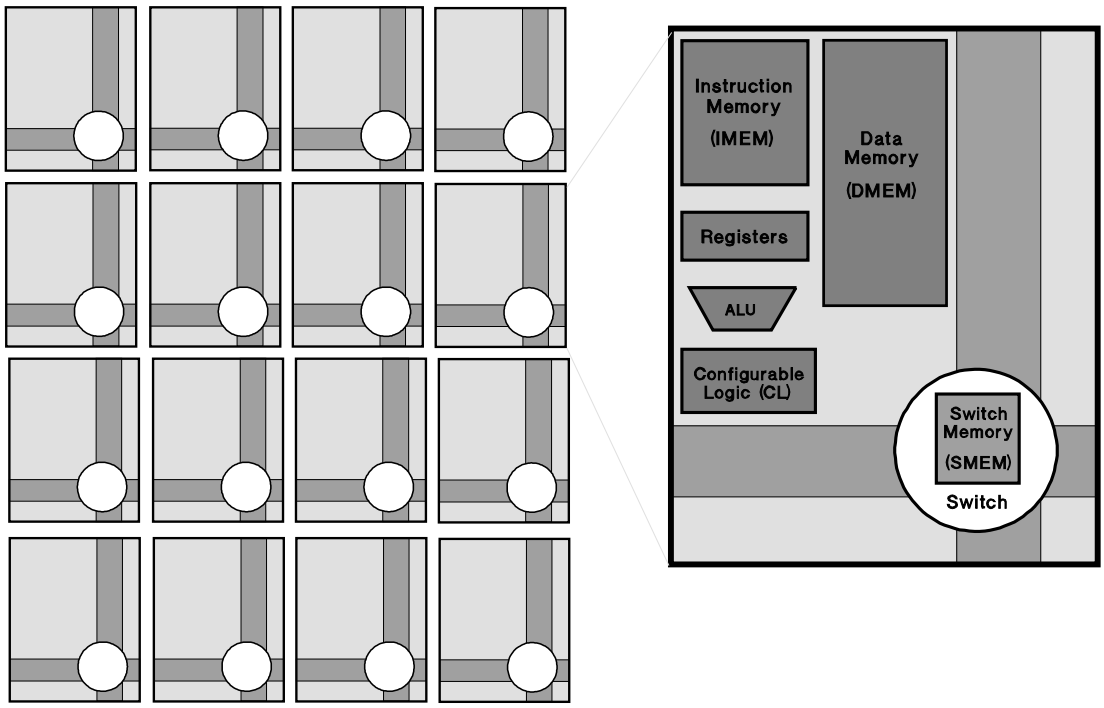


Figure 2.8: Architecture of the RAW Architecture, showing a portion of the tile array, and the details of one of the tiles.

processed by the tile. Another way of viewing this intimate relationship of the configurable logic to the ALU is that it allows the compiler to generate new specialized instructions, and insert them into a data stream that is being processed by the "normal" instructions built into the ALU.

The RAW Architecture is extremely versatile. The additional of the programmable network in the chip provides the capability of systolic-array architectures. The ability to process several parts of a data stream through different tiles, with either the same program (as in Intel MMX-type (Peleg, Wilkie, and Weiser 1997) instructions) or different ones provides a VLIW-like capability. If the ALU does NOPs, so that the data stream is processed entirely by the reconfigurable section, a fine grained FPGA-like capability results. In fact, the RAW architecture contains characteristics of virtually every processor configuration, allowing it to emulate any of these efficiently.

The downside of such a versatile architecture is the heavy burden placed on the programmer and/or compiler to effectively utilize the available resources. As a result, a

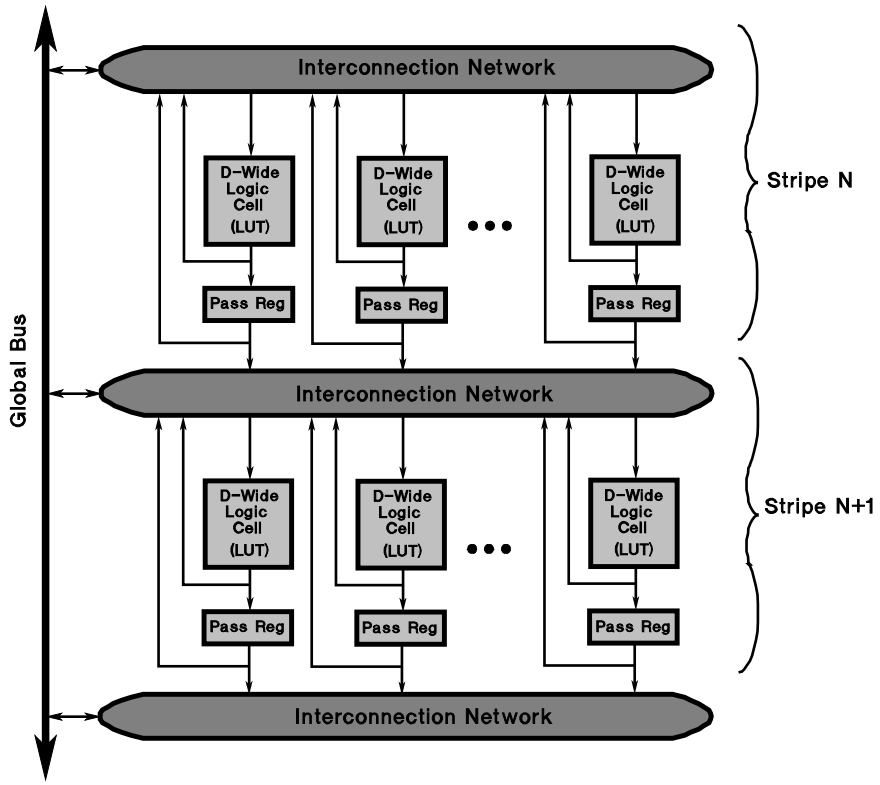


Figure 2.9: Architecture of the PipeRench Architecture, showing two stripes.

significant effort is being placed on the compilation system to go along with the hardware development; some aspects of the compilation process is discussed later. Also, since the entire system's throughput depends upon properly balancing the amount of chip space devoted to the various components within each tile, extensive simulations are being done to determine the optimal sizing. To date, no actual implementations have been completed, although a detailed specification of the prototype has been developed(Taylor 2004).

2.3.3 PipeRench

In contrast to the chunky architectures described above, the PipeRench Project (Goldstein, Schmit, Moe, Budiu, Cadambi, Taylor, and Laufer 1999) is more fine grained in nature, although not as fine grained as netlist architectures. The main characteristics of the PipeRench architecture are shown in Figure 2.9.

PipeRench is built around a pipeline scheme – each pipeline stage is called a *stripe*. Several stripes are connected together, with the top and bottom connected together to

form a ring. The set of stripes in a system is called the *fabric* - the exact number of stripes in a given system is implementation dependent. Each stripe contains a number of small reconfigurable (lookup table based) PEs. The usual situation is for the inputs of one PE to come from the outputs produced by the previous stripe, and for the outputs of the PE to be connected to the inputs of the next stripe. However, a set of global busses along each edge of the fabric also allow non-contiguous stripes to communicate with one another.

A PipeRench program is written in a single-assignment, pipeline oriented language called DIL (Goldstein and Budiu 1999). The compiler translates the program into a set of pipeline stages; at the same it does the synthesis and place-and-route operations. This is in contrast to other netlist-based compilation systems, which may require hours to perform these tasks. It is possible to do in reasonable time here because of the relatively simple interface structure between the stripes. The execution behavior is rather novel; the run time part of the compiler maps the first n ($n <=$ the number of stripes) pipeline stages onto the PipeRench fabric. Since it is likely that more pipeline stages are required than the fabric contains, once a stripe has been executed, it is replaced with another one; the effect is a "rolling" execution through a circular buffer of stripes, like the tread on an army tank, with stripes that have just executed being reprogrammed with code that will soon be executed.

At first glance, it would seem that this process would be inefficient, since it is possible that a stripe that occurs in a long loop (one that has more pipeline stages than the fabric) will need to be loaded multiple times. This is true; however, the loading of new stripes occurs while other stripes are executing; thus, the cost of the reconfiguration can be completely hidden by the execution time of the currently loaded stripes.

Several things are interesting about the PipeRench architecture:

- The compiler required to translate DIL into stripes is simple. Unlike some other architectures, which hope that their compilers can handle a very unwieldy task in producing efficient code for a complicated architecture, the DIL compiler is simple and straight-forward.

- Because of the simplicity of the interface between stripes, the compiler also performs

the synthesis and place-and-route. This eliminates some of the most time-consuming steps involved in producing reconfigurable programs.

- The architecture is extremely extensible. A code can operate on a fabric with any number of stripes. If a larger fabric is available, the same program will operate without recompilation on the larger fabric, with the benefit that more of the complete program will fit on the fabric, thereby reducing the amount of context switching required. The width of stripes (i.e., the number of data bits in each pipeline stage) can also be changed, although this requires a recompilation for the new width.

A first implementation has been fabricated, and it currently being tested. A second second iteration on the design has started (Goldstein and Schmit 1999), with a focus on enhancing I/O performance, and to provide a shared memory capability between stripes. Also, a piperench chip has been fabricated and tested, using 0.18 micron technology(Schmit, Whelihan, Tsai, Moe, Levine, and Taylor 2002).

2.3.4 Field Programmable Processor Array

Another architecture whose granularity falls between the very fine-grained netlist and coarse-grained chunky architectures is called the Field Programmable Processor Array (FPPA)(Donohoe, Buehler, and Bruder 2000b). It is being implemented using radiation-tolerant design techniques, so that it can be used in satellite and other outer space applications.

An overview of the architecture is shown in Figure 2.10. It consists of sixteen identical *Processing Elements* (PEs), organized into four groups of four called *clusters*. The switch within the cluster allows any PE output to be connected to any other PE input with the cluster. A central crossbar allows each cluster can be connected either to other clusters, or to any of five Input/Output Ports (IOPs) or a Dedicated Output Port (DOP).

Each PE contains an ALU, multiplier, shifters and a comparator that can be connected in numerous ways to implement a function. The PE inputs are registers that can be updated at each clock pulse. In general, all datapaths are 16 bits; however, logic has been included so that the PEs in a cluster can be connected in parallel to create a wider

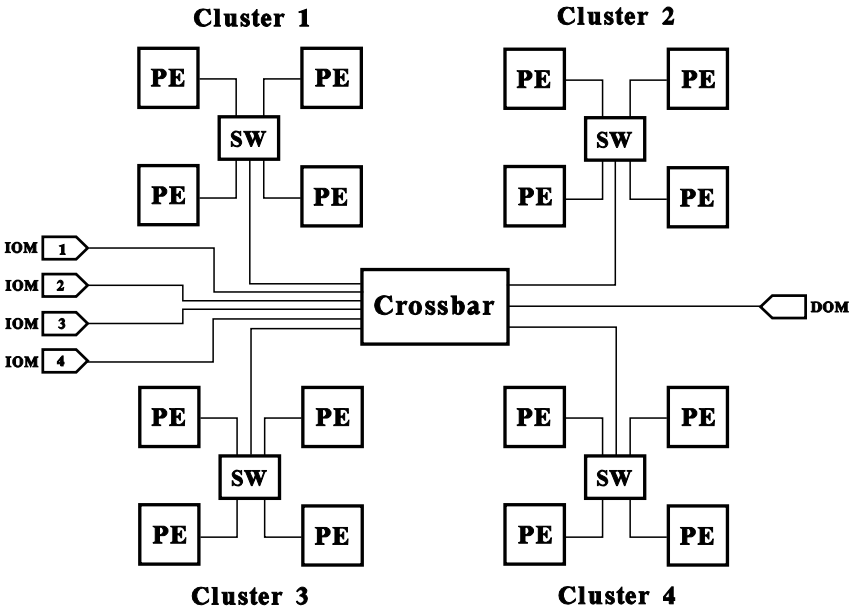


Figure 2.10: FPPA architecture overview

datapath when necessary. Figure 2.11 shows the simplified block diagram of each PE.

Due to its medium-grain design, a relatively small number of bits is required to fully specify the interconnection within a PE. A total of 70 bits specifies the interconnection of each PE. Similarly, the interconnect between clusters requires only 114 bits. A complete configuration of the FPPA therefore requires only about 1K (actually 1234) bits. Finally, each PE can be independently configured; it is not necessary to reconfigure the entire chip at once. Thus a typical application will generally require fewer than the maximum number of bits.

At a minimum, an FPPA "program" consists of a configuration that specifies the interconnection within the PEs and between PEs, and a "runtime sequence" that specifies when new values are to be loaded into the input registers of each PE. A program "executes" by first loading a configuration into the PEs, and then executing the runtime sequence to determine when, and which, input registers are updated. For example, several PEs might be connected in series to create a pipeline; in such a configuration, a pipeline stage consists of the PEs input registers, followed by the interconnected logic within the PE. Alternately, the PEs in a cluster can be connected in parallel to accommodate data paths that are wider than 16 bits.

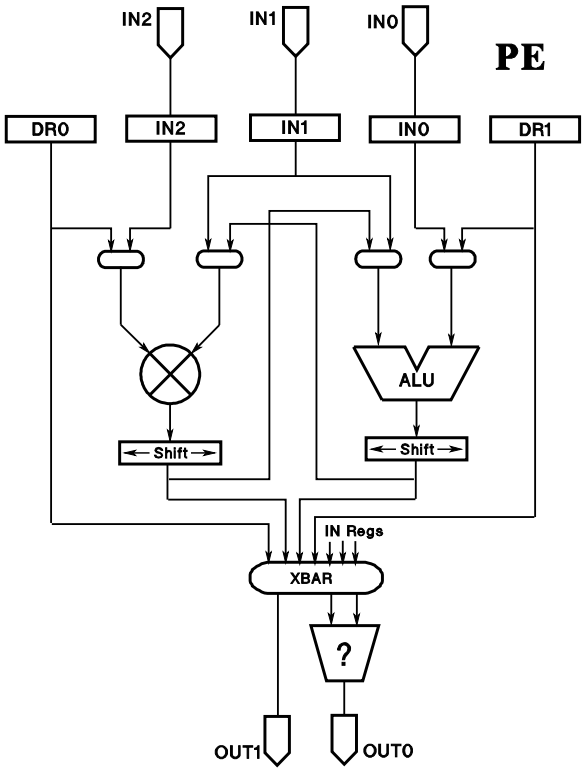


Figure 2.11: Organization of each FPPA Processing Element

Several applications have been coded for the FPPA and currently run in simulation (Sabde, Buehler, and Donohoe 2003). The architecture has been influenced by the requirements for these applications, as well as an effort to develop a fixed-point arithmetic capability for the processor(Buehler, Donohoe, and Yeh 2004; Buehler 2004). Of particular note for the subject of this dissertation, an effort is being initiated to re-target the SA-C compiler for the FPPA(Rinker, Nathan, Buehler, and McConaghy 2005).

2.4 Software Development for Reconfigurable Computing

Perhaps even more important than the novel hardware being developed for reconfigurable computing is the challenge of software support. The best, fastest hardware will not be utilized unless software exists that can exploit the hardware resources.

In a look ahead at the future of reconfigurable systems, Hauck (Hauck 1998) states that:

"...if reconfigurable computing is to be successful it must create a methodology

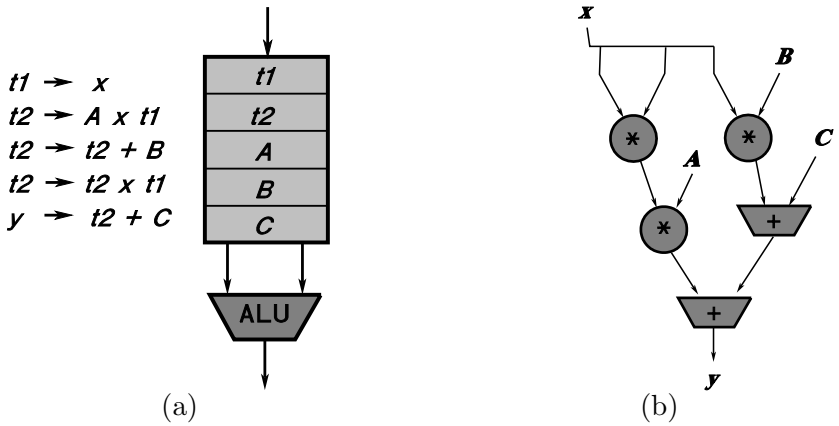


Figure 2.12: A comparison of (a) temporal vs. (b) spatial computation for the equation $y = Ax^2 + Bx + C$

> *to automatically map from standard programming languages to the hardware system. Just as general-purpose computing has moved away from assembly language programming, future users of reconfigurable computing will either insist on such a mapping tool suite, or will abandon this technique."*

This is a daunting challenge, due to the significant differences in the execution behavior of the target machine for reconfigurable computing versus general processors. This difference is illustrated in Figure 2.12.

General-purpose sequential processors perform *temporal computation* (DeHon and Wawrzynek 1999) – a set of instructions is executed in sequence over time. In its simplest form, one instruction is in execution at any instance, and the order of instructions is fixed. A clock determines the point at which one instruction is finished executing and the next instruction starts. In contrast, reconfigurable computing exhibits *spatial computation*, where the computation is performed by a uniquely connected circuit, and is able to proceed as quickly as the underlying technology allows. This new execution model motivates several important features of the nature of reconfigurable computing:

- The computation is no longer bound by a clock and a sequential ordering of instructions. Instead, the order of operations involved in the computation is defined by the interconnection pattern in the computation circuit. This behavior, in the context of digital logic, is called a *combinational circuit*, or a combinational computation. The "answer" for a given set of inputs appears at the bottom as quickly as the circuits

can respond to that input (i.e., the propagation delay of the circuit), rather than being regulated by a clock.

- Spatial computation is naturally parallel, since all parts of the circuit are "computing" at the same time.

- The computation can be characterized by a flow of information from the top of the computational circuit down to the bottom – the speed and order with which the computation proceeds is determined entirely by the interconnections of the circuit and the propagation delay inherent in the implementing technology. This characterization is embodied in the *dataflow graph* described earlier – reconfigurable computing appears to be a nearly perfect practical implementation of the abstract dataflow execution model.

A very important feature of temporal computing - that of loops, does not exist in purely spatial computing. The effect of loops is produced in one of two ways:

1. By unrolling the loop, which transforms the temporal concept of an iteration to an entirely spatial entity, or,

2. By introducing a *state machine* which controls the generation of input values, which are presented to the inner loop body as a data stream. The circuit is no longer strictly combinational, since the state machine introduces the notion of time back into the computation. However, the execution behavior is still different from the traditional processor in that the inner loop body is allowed to compute as quickly as the circuits can respond, and the operation remains inherently parallel. The speed at which the computation proceeds is limited by the propagation delay of the inner loop circuits.

Standard languages, such as C, are designed to represent a temporal style of computing – as such, they are ill-suited for expressing spatial computing. A considerable amount of effort must be spent to determine unnecessary sequentialness specified by the language and to expose the parallelism that exists in a calculation, so that a spatial representation

of the problem can be formed. Often this involves severely limiting the subset of the language that can be used for reconfigurable computing. Nonetheless, there is considerable motivation to start with a standard language, since it is familiar to the application programmers who might effectively utilize reconfigurable computing.

The next sections provide an overview of several software-related research efforts in reconfigurable computing.

2.4.1 The RAW Compiler Project

Coupled with the RAW hardware project discussed earlier is a significant compiler effort (Agarwal, Amarasinghe, Barua, Frank, Lee, Sarkar, and Srikrishna 1997). The goal is to provide a C compiler which targets the RAW architecture. In some respects, the compiler is fairly conventional, since the components that make up each tile of the architecture (a RISC CPU with data and instruction memory, a regular interconnection network, and an intelligent data switch) are relatively conventional. It is unique, however, in that all of the resources of RAW reside on a single chip, and therefore the compiler can assume a very regular array of resources. This is in contrast to other networked resources such as WAN or networks of workstations, where the interconnection network is irregular (both in interconnection and in communication latency) and changeable, and communication costs are high. The compiler can exploit this regularity, for example by "ganging" several tiles together to perform a highly parallel or wide datapath operation. This produces the effect of a VLIW machine with a variable-width instruction tailored to a specific application, and with the further flexibility that the processors do not have to stay in lock-step during execution.

Similar flexibility exists in the scheduling of communication events. With a regular, low-cost communication network that contains intelligent switches, both static and dynamic routing can be accommodated simultaneously.

The current compiler implementation is called RAWCC, and it compiles both C and Fortran Programs (Lee, Barua, Frank, Srikrishna, Babb, Sarkar, and Amarasinghe 1998). It operates in three phases:

- Traditional high-level program analysis and transformation, including memory disambiguation, loop unrolling, and a scheme which encapsulates the control flow within a basic block into a single unit for resource scheduling purposes.
- A basic block orchestrator, which decomposes a single basic block into a set of basic blocks that can be executed in parallel on separate tiles. This involves both time and space scheduling of each basic block on a set of tiles.
- Code generation for the processors and switches.

Results show that the speedup ranges on common benchmarks from 9 to 38 on a 32 processor RAW simulator, as compared to a single processor. The superscalar speedup occurs due to scalar optimizations exposed during loop unrolling, a phenomenon more related to the highly parallel nature of the code than on inherent qualities within the RAW architecture.

2.4.2 PipeRench and DIL

The PipeRench architecture described earlier is programmed with a language called DIL (*Dataflow Intermediate Language*) (Goldstein and Budiu 1999). DIL is intended to be an intermediate language between a high level language such as C and the PipeRench hardware description. DIL is designed specifically to describe the interconnections possible within stripes of the PipeRench architecture – the language and the architecture are a perfect match for each other. Because of this, DIL can be compiled almost directly into an interconnect specification for PipeRench; very little effort is required in place and route operations after compilation.

DIL is a single assignment language - a variable can be assigned a value only once. Thus, there is a one-to-one correspondence between variables in DIL and wires in the hardware, one of the reasons for the close tie between language and hardware. The delays required between expression evaluation and variable assignment is also explicit; for example:

```
var <n= expression;
```

indicates that `var` is to be assigned the value of `expression` after n clock cycles. This statement is implemented by using one stripe to calculate the expression, and then connecting the output of the stripe (possibly via the global bus) to the input of the n-th stripe following it.

Different from C, data types have no inherent size (i.e., bit width) associated with them – data sizes are either specified by the user, or are inferred by the compiler from the input and/or output sizes (the user must specify these). This owes from the fact that reconfigurable hardware (including PipeRench) usually has no inherent bit size restrictions, except to have some maximum size determined by I/O or other architectural interfaces. In particular, considerable size and sometimes speed efficiency can be gained by using minimum bit widths for calculations.

DIL is intriguing due to its simplicity, and to its close correspondence to the hardware which is its target. It is also dataflow-like in nature, which means that, within the limitations of high-level languages to express parallel operations, it should be relatively straight-forward to translate these high level languages to DIL.

2.4.3 Reconfigurable Computing and High Level Application Environments

A couple of research projects which use higher level application environments as a front-end for reconfigurable computing are ongoing. The first, called MATCH (Banerjee, Shenoy, Choudhary, Hauck, Bachmann, Chang, Haldar, Joisha, Jones, Kanhare, Nayak, Periyacheri, and Walkden 1999; Periyayacheri, Nayak, Jones, Shenoy, Choudhary, and Banerjee 1999), involves the use of MATLAB as its input language. The project is developing a MATLAB compiler and a set of library modules that implement the various MATLAB functions. They have reported a two orders of magnitude increase in computation rate of certain MATLAB benchmarks over the C versions of the libraries.

The second project, called CHAMPION (Natarajan, Levine, Tan, Newport, and Bouldin 1999), involves a similar approach, except it uses Khoros as the input language. Khoros glyphs are written VHDL, and then they are mapped onto hardware. This project is of particular interest here, since the problem description is very similar to that of

Cameron, at least on the surface, and the two projects are currently targeting the same hardware. The main difference is that the Khoros glyphs are written in VHDL in Champion, and in SA-C in Cameron. The focus of effort in Champion has been on the rewriting of the glyphs in VHDL, and the partitioning and mapping of this code onto the hardware, whereas in Cameron the effort has been focused on compiler development and automation of the development process, in order to provide an environment in which to develop glyphs that run on hardware. The orthogonal approaches between the two projects suggest that the two could benefit from collaborating with one another – Champion could utilize the SA-C compiler to develop their glyph code, and Cameron could take advantage of the problem partitioning that has been developed by Champion.

2.5 The Cameron Project and SA-C

The Cameron Project (Najjar, Draper, Böhm, and Beveridge 1998; Hammes, Rinker, Böhm, Najjar, Draper, and Beveridge 1999) is a software project whose goal is to provide a hardware-independent environment for reconfigurable computing application development. Its primary application domain is image processing, although the environment is general enough to be suitable for other areas as well. Since the research described in this dissertation is being performed in conjunction with the Cameron project, its description has been saved for last. This section also serves as an introduction to the SA-C language and compilation system that will be described in more detail later.

Central to the project is the design of a language called SA-C (*Single Assignment C*, pronounced *"sassy"*) (Hammes, Draper, and Böhm 1999; Hammes and Böhm 1999). It consists of an optimizing compiler which produces as its intermediate form a dataflow graph (DFG), a translator which converts the DFG into VHDL, and some ancillary programs which aid in debugging and program verification. As the name implies, SA-C is a derivative of C – the language shares its operator syntax and precedence, and the available data types have been extended from C. However, many of the features of C which would require considerable analysis for reconfigurable computing targets have been removed or modified.

SA-C shares some similarities to DIL, due to their common heritage as hardware generation languages. Like DIL, SA-C is single assignment, and uses both user specification and size inference to try to minimize the bit widths required for calculations. In contrast to DIL, SA-C is a considerably higher level language, and is designed to be independent of the particular hardware that is used (indeed, SA-C can be compiled to run only on the host).

One important feature of the language, different from C, is *loop generators*. In most languages, a particular array access pattern is specified by creating one or more explicit loops to generate indices that are subsequently used as array subscripts. In order to perform any memory access optimizations, the compiler must recognize the interrelationship between these two separate operations. By contrast, SA-C combines these operations into a single loop generator function; the compiler is free to choose an efficient hardware implementation which produces a "stream" of properly–ordered data that is presented to the inner loop body (ILB) to perform the loop's calculations. The existence of true n-dimensional arrays allows the compiler infer the "shape" of the result. Generators which extract *windows* (small $m \times n$ sub–arrays) of data from larger arrays are also provided.

The language also contains *reduction* operators – language primitives which perform commonly used operations on the data produced by the loop bodies. Some reductions are simple operations such as *array* (i.e., create an array from the result values) and *sum*; others perform more complicated functions such as *histogram* and *median*. The reduction operators eliminate the need to express these algorithms specifically in SA-C, thus allowing the compiler to use efficient VHDL (hardware) implementations.

For hardware generation, the primary output of the SA-C compiler is a DFG, which serves as the input to the hardware translator. This graph is flat (not hierarchical), acyclic, and contains no artificial sequential dependencies – the nodes of the DFG are related to each other only by the true data dependencies that exist in the original program. It contains operators that can be translated into VHDL. All of the parallelism that exists in a program is exposed in the DFG; this facilitates the hardware translation.

The compiler also generates other outputs. Along with the DFG, the compiler produces

the host-based driver program. This program, written in C, initializes the runtime system, performs the non-RCS functions of the SA-C program (such as file I/O), downloads the configuration codes and image data to the RCS, and initiates RCS execution. Optionally, the compiler can produce a *C-dump*, which is a stand-alone C program which is the functional equivalent of the input SA-C program. It allows the functional verification of the program without having to go through the time-consuming process of hardware generation, and can execute on a host which is not connected to the RCS hardware.

The SA-C language has been integrated into the Khoros design environment (Konstantinides and Rasure 1994). Khoros *glyphs* can be written SA-C, tested and executed on the host computer using Khoros, and then converted to hardware. Khoros can then be used to execute the final application on the hardware.

Considerably more detail about SA-C, its optimizations, and in particular the translation from the intermediate Dataflow graphs form to VHDL (the focus of this dissertation) will be presented in the next chapter.

2.6 Summary

The topic of reconfigurable computing has been an active area of research over the past several years. Both hardware and software projects have been involved. It appears that new research in the area is beginning to wane, now that the benefits and limitations of reconfigurable computing are beginning to be understood. Reconfigurable computing is beginning to become a mature technology; it can take its place among other technologies as a possible solution to problems requiring performance in certain specific applications, especially those involving parallel, repetitive computations.

Chapter 3

A Preliminary Project - Manual Implementation of an RCS Design

The development of the SA-C compilation system occurred in two main phases, roughly corresponding to the "front-end" and "back-end" components of the system. The first phase of the project involved the specification of the SA-C language and an initial implementation of a compiler for the language. This compiler produces two primary outputs. The first is a host-only executable, called a *C-dump*, which was used by the compiler designers to verify proper operation of the compiler before the RCS portion of the project was implemented. The second is the DFG, which is used as the input to the second phase of the project - producing the RCS code. As a result of some superior compiler expertise and incredible effort on the compiler team, a very robust compiler has been available from the early stages of the project.

The second phase – that of translating DFGs to hardware, the primary subject of this dissertation – was much more of an unknown process. In an attempt to better understand this problem and its challenges, a manual, non-automated implementation of of a common Image Processing (IP) algorithm, the Prewitt edge detector, was designed by this author and implemented on the AMS Wildforce-XL RCS board. This chapter describes the results of this exercise. This effort was fruitful in several ways:

- It allowed the hardware-oriented part of the team to become familiar with the RCS system, and the VHDL language that is used to program it.

- It allowed practice with the development environment used for RCS's, and develop-

ment of debugging techniques.

- It allowed the team to explore alternative solutions, particularly from a performance standpoint.
- It helped identify numerous issues involving performance optimizations that were later incorporated into both the compiler and the DFG translator, and
- The resulting design serves as a performance benchmark for the automated system that is produced, much like a program, manually written in assembly language, might serve as a benchmark to evaluate the performance of the same program written in a high level language.

3.1 Background - the Prewitt algorithm

The Prewitt algorithm is a well known edge-detection algorithm used in many IP applications; its development and operation is described in (Prewitt 1970), and in most image processing texts, such as (Gonzalez and Woods 1992). The actual algorithm involves the convolution of an image with two constant 3×3 masks – one which forms the X gradient, and one which forms the Y gradient; the two masks are shown in Figure 3.1.

$$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{pmatrix} \qquad \begin{pmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{pmatrix}$$

(a) (b)

Figure 3.1: Constant convolution masks for the Prewitt algorithm. (a) the X gradient mask, and (b) the Y gradient mask.

The results of these convolutions form two vector components; the magnitude of the resulting vector forms the desired result. This fundamental operation is performed over the entire image, with the results being another $2D$ array called the *gradient array.*

While this algorithm is important in its own right as a fundamental IP operation, it was selected for this exercise for other characteristics that make it an interesting example for Reconfigurable Computing:

- It is inherently an extremely parallel operation, with each 3×3 convolution being entirely independent of the other calculations.
- It involves the use of constant masks, which allows for considerable optimization before being implemented in hardware.
- It presents some computational challenges since it requires a squaring (multiplication) and square root, which are difficult on FPGAs.
- It involves the use of a streaming data model from host to RCS back to host, which appears to be a common mode of operation in IP algorithms.
- It is representative of a large number (perhaps even the majority) of other common IP applications. For instance, the design of the SA-C language is built around the concept of *window generators*, which provide a simple means for expressing the extraction of a small sub-array from a larger one.
- The performance of this manual version will serve as a benchmark with which to compare the automatically generated version.

3.2 The Wildforce-XL Board Architecture

The Wildforce-XL board (Annapolis Micro Systems, Inc., Annapolis, MD 1997) was described earlier in Section 2.2.2 - its architecture is pictured in Figure 2.4. It contains five user-programmable FPGA chips; the specific board used in this project is populated with five Xilinx 4036XL components, each containing a matrix of 36×36 (1296) CLBs. The resources of the board allows considerable flexibility in choosing a model for a particular application. For the Prewitt algorithm, CPE0 (the "control processor") receives image data (8-bit pixels) from the host processor over the PCI bus. It distributes this data to the other FPGAs, PE1 thru PE4, via the *crossbar*, which is configured so that CPE0 can broadcast data to all the PEs at once. PEs 1–4 are nearly identical in function. Each PE is connected to its left and right neighbors via a 36 bit bidirectional bus. The end PEs (1 and 4) are connected to a bidirectional FIFO, instead of to a neighbor, which can be used

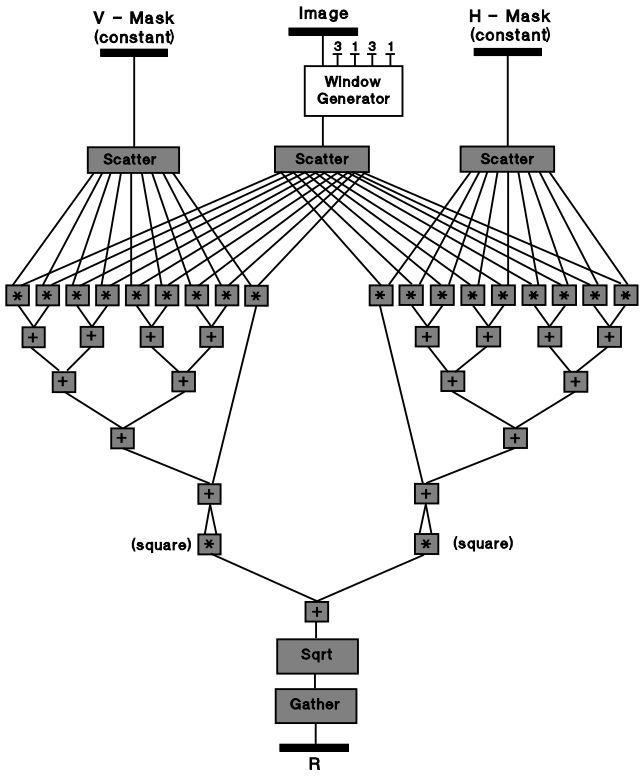


Figure 3.2: Dataflow graph of the unoptimized Prewitt algorithm inner loop body

to communicate data to or from the host. In addition, the PEs each have a signal line (not shown in the figure) that is used to coordinate activities with CPE0. Finally, each PE has a local memory (512K bytes, organized in 32-bit words) which can be accessed by both the PE and the host processor.

3.3 The Inner Loop Body of the Prewitt Algorithm

The first step in implementing the algorithm is the development of the inner loop body (ILB), that is, the convolution calculation that is the core of the algorithm. For each convolution operation, one of the 3×3 sub-matrices in the image is convolved with the two constant masks. The resulting values represent the X and Y components of the gradient vector. The magnitude calculation involves the squaring of each gradient component, summing these squares, and then finding the resultant square root. A dataflow graph of this entire operation is shown in Figure 3.2. Note that before being optimized, the inner loop body requires 17 additions and 20 multiplications, along with the square root

computation. This amount of calculation is prohibitive in both time and space on an FPGA.

Fortunately, considerable optimization can be performed on the ILB before committing it to hardware. While for this exercise this optimization was performed by hand, it is expected that the SA-C compiler will be able to perform most, if not all, of these optimizations; indeed, the results currently being obtained from the compiler are very close to the manually derived ones. The optimizations include:

- The first line of multiplies all involve small constants (either zeroes or ± 1); they can be eliminated or replaced with simple additions or subtractions.

- A naive view of the algorithm would reveal that 10 bits are necessary for representing the gradient values. However, in practice the values are usually small, representable in 5 bits or less. Larger values simply indicate stronger edges; little information is lost by allowing the values to saturate to 31 (the largest 5-bit number). This allows the subsequent squaring and square root operations to be reduced in width.

- The bottom of the DFG in Figure 3.2 computes the magnitude of two gradient components; this operation is rather slow. The operation can be replaced with a lookup table – for 5 bit numbers this table is 1024 entries long, so easily fits in the PE local memories. The lookup table reduces the propagation delay of the ILB by nearly 75%, at the expense of the two cycle memory access to do the lookup.

- The propagation delay can be further reduced by adding pipeline registers to the remaining part of the ILB. Adding these registers allows the system to operate approximately 70% faster than the non-pipelined version.

After optimizations, the dataflow graph for the algorithm is shown in Figure 3.3. The hardware implementation is able to run at speeds in excess of 10 MHz for the non-pipelined version, almost 17 MHz for the pipelined one.

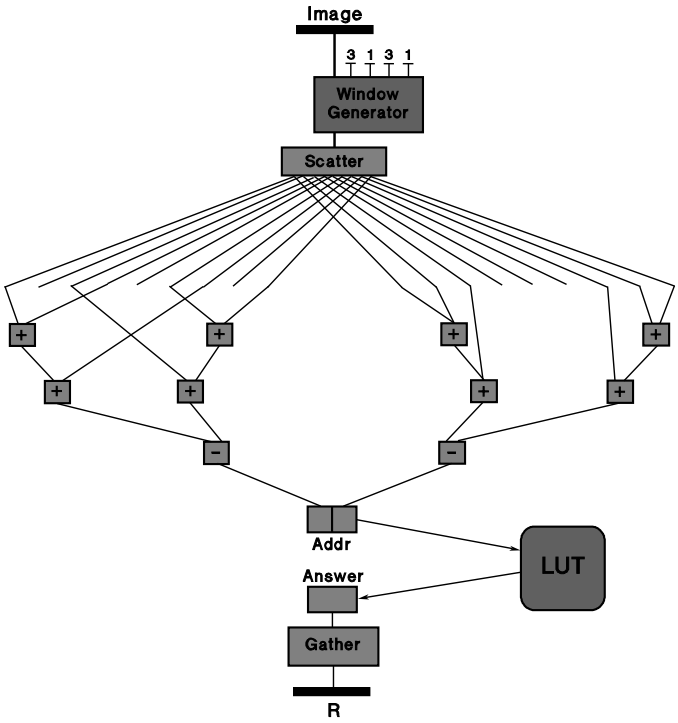


Figure 3.3: Dataflow graph of the optimized Prewitt algorithm inner loop body

3.4 Distribution of Image Data from Host to the PEs

The Wildforce board provides two main ways for transferring data between the host and board – via dedicated First-In-First-Out (FIFO) buffers built into the processor, and through block transfers by the host to the PE memories. Both techniques were evaluated during this project.

In both versions, CPE0 is responsible for accepting data from the host, and the other PEs are responsible for performing the actual calculations. They differ in the manner in which data is handled through the board. In both versions, there is enough space in each PE to instantiate more than one convolution calculation at a time, so multiple instances can operate in parallel.

The order in which data is sent influences the efficiency of the calculation. In order to perform a Prewitt calculation, a PE needs to receive a 3×3 sub-matrix of 8 bit pixels. If this data is sent one column (3 bytes) at each clock cycle, after three clock cycles a single convolution calculation can be performed. This scheme is shown in Figure 3.4(a),

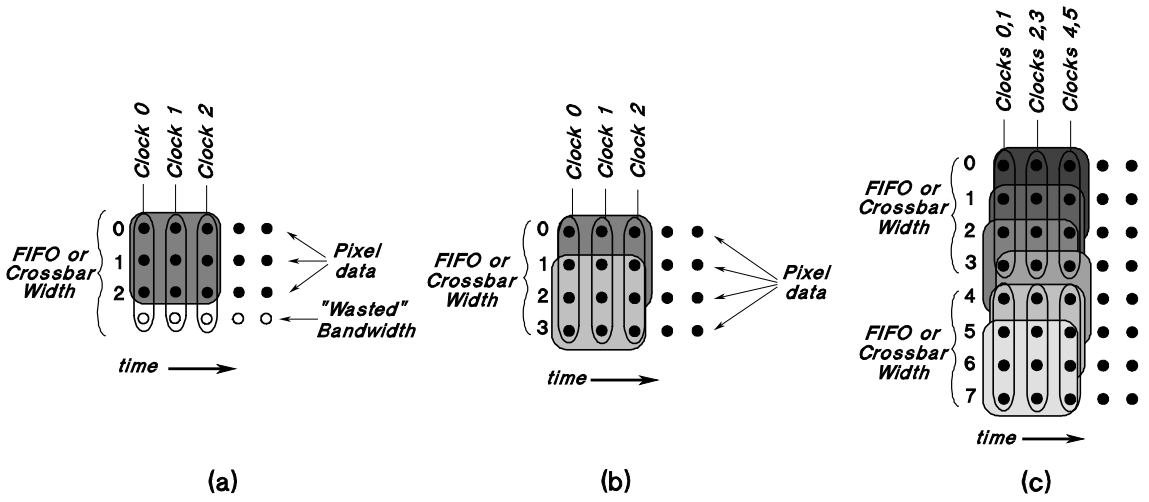


Figure 3.4: Three data transfer schemes: (a) Single instance of the convolution calculation, (b) two instances, and (c) six instances

where pixels are represented by the black dots, the data transferred at each clock cycle is shown by the ovals (i.e., time moves from left to right), and the shaded square shows the pixels involved in the convolution calculation. Since the datapath is 36 bits wide, several bits of bandwidth are wasted at each cycle, represented by the empty circles. However, if *four* elements of each column are transferred at each cycle instead, not only will the bandwidth be more fully utilized, but in three clock cycles the PE will receive enough data to calculate *two* Prewitt values, thereby doubling the rate of calculation. This scheme is shown in Figure 3.4(b). Since the two Prewitt calculations can be performed in parallel, this additional performance is achieved with no additional time cost. The inner loop of the Prewitt computation must be replicated twice instead of once; however, this extra replication requires only about 10% of additional space.

Taking this concept one step further, yet another variation in sending data is to extend the convolution calculation instances from two to six. Now, two clock cycles are required to send enough data to supply all six instances, resulting in three times the calculation throughput; this is shown in Figure 3.4(c). This scheme requires six instantiations of the inner loop body on each PE and a somewhat more complicated wiring scheme. The resulting circuit occupies more than 80% of the FPGA, which is near the "comfortable" capacity of the FPGA. Nonetheless, this exercise points to the ability of FPGA-based

computing to exploit parallelism in an algorithm.

Thus, with two instances, the data rate required to provide two calculations can be sustained without adding clock cycles for the transfer. For more than that, one additional cycle is required for data transfer to supply up to six instances of the convolution. Since six instantiations also consumes most of the space in each PE, it represents the maximum parallelism that can be implemented on the Wildforce boards. Both two and six instantiation versions were implemented for the two data delivery methods.

It was first thought that the FIFO version would be the better choice, since the FIFO operation seemed to best match the "streaming" model that seemed appropriate for this application. The host can be sending data to an input FIFO while the Wildforce board is simultaneously retrieving the data; this helps to hide the transmission latencies between host and board. As the PEs produce results, they are sent, in "systolic" fashion, across the interconnection network between PEs to one or both FIFOs on each end of the PE array. Again, the host can be retrieving data simultaneously from the output FIFOs. The data rates between the various parts can be regulated easily by the host – it sends more data to the input FIFO only after it has retrieved enough data from the output FIFOs to prevent the new data from overrunning the FIFO. Another important aspect of this design is that the host handles the responsibility of sending the data in the proper order to the Wildforce board; in other words, the host implements the window generator function by determining the order of data.

Three variants using the FIFOs were implemented. The simplest one used a single FIFO (FIFO4 in Figure 2.4) for output. Another used a single output FIFO, but pipelined the calculation for more speed. The third used two FIFOs (FIFO1 and FIFO4) for output.

The performance results using the hardware FIFOs is summarized in Table 3.1. Some explanation of the results is in order. The design is first expressed in VHDL, then the design is presented to a synthesis tool (Synplify, in this case), which generates a netlist in terms of CLB components – registers, F-Maps, and H-maps. This netlist is then processed by several place-and-route stages (the Xilinx M1 tools) to produce a final design. The place-and-route tools often must add resources to the design, therefore the results differ

CPE0 - FIFO Input

	Pre-PAR (Synplify)					Post-PAR (Xilinx M1)				
Design	Regs	FMAPs	HMAPs	Total CLBs	Freq MHz	Regs	FMAPs	HMAPs	Total CLBs	Freq MHz
2 Prewitt	75	62	7	38 (3%)	52.1	80	63	8	67 (5%)	31.3
6 Prewitt	76	64	8	38 (3%)	57.1	81	65	9	68 (5%)	26.5

PE1-4 – FIFO Output

	Pre-PAR (Synplify)					Post-PAR (Xilinx M1)				
Design	Regs	F MAPs	H MAPs	Total CLBs	Freq MHz	Regs	F MAPs	H MAPs	Total CLBs	Freq MHz
2 Prewitt										
1 FIFO Out	297	403	12	202 (16%)	19.0	293	404	14	430 (33%)	11.3
1 FIFO Pipe	337	413	26	207 (16%)	26.8	333	414	28	435 (33%)	16.9
2 FIFO Out	297	401	10	201 (16%)	19.0	293	402	11	431 (33%)	10.6
6 Prewitt										
1 FIFO Out	564	1103	80	552 (43%)	17.2	559	1104	82	1087 (84%)	10.3
2 FIFO Out	527	1052	36	526 (41%)	17.2	525	1050	39	1042 (80%)	10.4

Table 3.1: Results in terms of space (CLB usage) and maximum clock rate for pre-Place and Route (as reported by the Synplify VHDL compiler), and post-Place and Route (reported by the Xilinx M1 tools) for several of the FIFO-based designs described in this report.

between synthesis estimates and the final design – both are reported in Table 3.1. The percentages represent the percent of the total amount of space on each FPGA required by the design. As a rule of thumb, utilization of 75% to 80% is the practical limit on capacity – utilization beyond that often causes the place-and-route process to fail.

A separate configuration is created for CPE0 and the other PEs - the overall clock speed of the Wildforce board is determined by the slowest of the two configurations. In all versions, the speed of the PE configuration determined the overall clock speed.

The FIFO results appear to be somewhat disappointing:

- A considerable amount of data has to be re-transmitted between host and the control processor CPE0, because nearly every pixel is involved in more than one window of data. This places pressure on the host-to-board bandwidth, which is the system bottleneck even without the re-transmitting.

- The host-to-FIFO transfer rate is very low. CPE0 was only able to keep the PEs about 7% busy, due to data starvation through the FIFO. The throughput through

the FIFOs appears to be very low. Even when a direct memory access (DMA) technique is used to transfer data between host and FIFO, the amount of data that can be transferred in each block in the transfer is relatively small (no larger than 512 bytes), so the overhead associated with setting up the DMA transfer is significant compared with the amount of data that is transferred.

- The process of sending data through the PEs to the FIFOs on the end turned out to be rather difficult. The PE on the end (next to the FIFO) must handle the data traffic from all PEs, and thus quickly becomes a bottleneck. Using both FIFOs for output helps to alleviate the problem, but nonetheless the problem still exists.

The second method involves transferring the entire image from the host to CPE0's memory at once. Then CPE0 retrieves the data from memory and distributes the data to the PEs via the crossbar, which perform the calculation and store the results in their memories. Finally, the host retrieves the results from the PE memories to complete the process. This set of operations must be done sequentially, because the memory design requires that only host or PE, but not both, can access memory; this means that transmission latencies cannot be hidden, and therefore add to the total processing time. On the other hand, with this design the window generator function resides on CPE0, rather than on the host; this eliminates having to send data more than once. And since the image data can be sent in large chunks, using DMA transfer, so that transfer rates are reasonable, up to 120 MBytes/sec, close to the PCI bus maximum of 133 MB/sec, and more than an order of magnitude faster than the FIFOs. Because of this much higher performance, this second method is the model chosen for all subsequent experiments. A picture of the memory-based system using two instances of the convolution calculations, and showing the various VHDL components that are replicated in each PE, is shown in Figure 3.5.

Table 3.2 shows FPGA space and Clock rate for the memory-based designs. Comparing these numbers with previous results from the FIFO version shows that while more space is required on CPE0, due to the fact that CPE0 is now doing the data distribution rather than the host, the amount of space used is still fairly small. On the other PEs, both speed

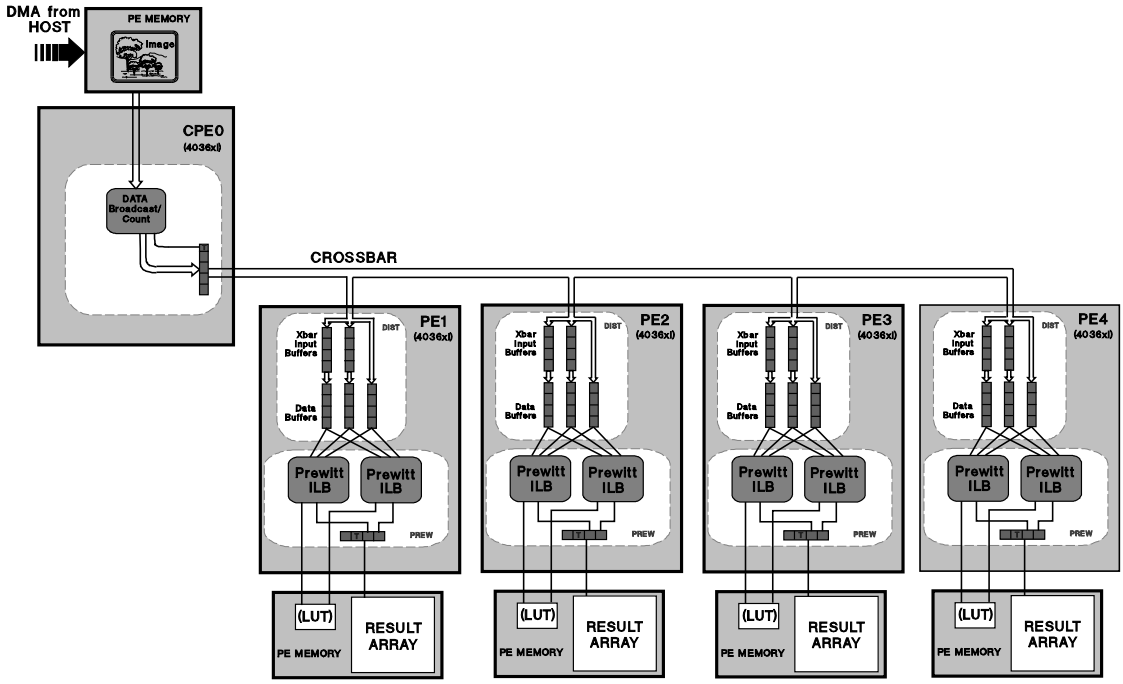


Figure 3.5: A pictorial representation of the entire Prewitt implementation, using two inner loop bodies per processing element

CPE0 - Memory (DMA)

	Pre-PAR (Synplify)					Post-PAR (Xilinx M1)				
Design	Regs	F MAPs	H MAPs	Total CLBs	Freq MHz	Regs	F MAPs	H MAPs	Total CLBs	Freq MHz
2 Prewitt	328	346	24	173 (13%)	26.5	331	347	70	338 (26%)	20.5
6 Prewitt	615	546	98	308 (24%)	23.0	618	547	131	717 (55%)	16.9

PE1-4 – Memory Output

	Pre-PAR (Synplify)					Post-PAR (Xilinx M1)				
Design	Regs	F MAPs	H MAPs	Total CLBs	Freq MHz	Regs	F MAPs	H MAPs	Total CLBs	Freq MHz
2 Prewitt	254	439	20	220 (17%)	18.8	253	441	28	420 (32%)	10.7
6 Prewitt	552	1081	25	541 (42%)	17.5	556	1082	68	1051 (81%)	10.7

Table 3.2: Results in terms of space (CLB usage) and maximum clock rate for pre-Place and Route (as reported by the Synplify VHDL compiler), and post-Place and Route (reported by the Xilinx M1 tools) for several of the memory-based designs described in this report.

Two - Prewitt Memory-based Design Performance

Design	Data Input	Prewitt Calculation	Data Output	Total Time	Overall Rate (MPixels/s)
320×240 **Image**					
Non Pipelined (10 MHz)	1.25 ms (60.5 MB/s)	9.25 ms (8.3 MPixels/s)	6.10 ms (25.18) MB/s	16.6 ms	4.6
Pipelined (17 MHz)	1.27 ms (60.9 MB/s)	6.03 ms (12.7 MPixels/s)	6.06 ms (25.35 MB/s)	13.34 ms	5.7
Pentium (C version)		35.07 ms		35.07 ms	2.19
512×512 **Image**					
Non Pipelined (10 MHz)	2.99 ms (87.7 MB/s)	23.01ms (11.4 MPixels/s)	8.24 ms (63.6 MB/s)	34.24 ms	7.65
Pipelined (17 MHz)	3.13 ms (83.7 MB/s)	13.1 ms (20.0 MPixels/s)	8.60 ms (60.9 MB/s)	24.83 ms	10.5
Pentium (C version)		107.0 ms		107.0 ms	2.45
$1k \times 1k$ **Image**					
Non Pipelined (10 MHz)	9.99 ms (104.9 MB/s)	83.9 ms (12.5 MPixels/s)	23.47 ms (89.3 MB/s)	117.36 ms	8.93
Pipelined (17 MHz)	10.3 ms (101.8 MB/s)	50.22 ms (20.9 MPixels/s)	23.02 ms (91.1 MB/s)	83.54 ms	12.53
Pentium (C version)		481.0 ms		481.0 ms	2.18

Table 3.3: Performance results for the memory designs, for several image sizes.

and space are comparable to the FIFO versions. However, now the system can be kept 100% busy (rather than only about 7% before), since the data is being retrieved from local memory, rather than being transferred to and from the host processor. There is an additional expense of the initial transfer of data to local memory, and transfer of results back to the host, but because of the large block transfers these are quite fast.

Table 3.3 shows the measured calculation rates for several images when using the 2-Prewitt memory design. This table also shows the measured results obtained by executing a C version of Prewitt on a 450 MHz Pentium host. It appears that the algorithm executing on the Wildforce board can be made to out-perform the same algorithm executing on more traditional hardware.

3.5 Conclusions Drawn from the Manual Prewitt Designs

Considerable insight was gained from the manual implementation of the Prewitt algorithm on the Wildforce board. These include:

- Separating the ILB of the application from the data generation function appears to be a reasonable approach to the design of a hardware application. Considerable speed and space efficiency can be gained by performing aggressive optimizations on the ILB before translating it to hardware. Since the ILB for each application is unique, the burden for the optimization will be placed on the compiler. The data generation function for each application can be accomplished by much more generic code.

- Transfers between host and RCS system can be slow; designs must try to use efficient methods in order to reduce this cost in time.

- The fastest way to transfer data between host and board is to use large block DMA transfer (larger than 256K bytes) between host and memory. We have measured transfer rates of up to 120 MB/sec with this mode, close to the theoretical PCI bus maximum of 132 MB/sec. However, even at this rate, the transfer of data between host and board often constitutes a significant fraction of the total time required for applications on the board. Our initial hunch that transmission time between host and board is corroborated by the results obtained in this project.

- The hardware FIFOs, which appear to be ideally suited for applications that require the streaming of data between host and hardware, seem to be useful only for low bandwidth transfers between host and board, around 8MB/sec maximum. The FIFOs support a DMA transfer mode; however, the block sizes are so small (512 words maximum) that the effective transfer rate still remains low.

- It is important for performance reasons to fully utilize the bandwidth of the hardware, to take advantage of the computational parallelism that can be implemented on the FPGAs.

These insights have been used to determine which of many compiler optimizations should be implemented in order to obtain the best performance. They also motivate some of the DFG optimizations that are discussed later.

Chapter 4

An Automated Translation from Compiler-Generated Dataflow Graphs to VHDL

With the insight gained from producing the manual implementation described in the previous chapter, attention turned toward implementing an automated process for converting DFGs into VHDL. This chapter describes the initial system that was implemented for doing this translation.

[1].

In order to put this part of the overall SA-C system into context, Figure 4.1 shows a diagram of the complete compilation process. The two main components of the process are the optimizing compiler itself, which produces an intermediate form called a dataflow graph (DFG), and the DFG-to-VHDL translator, which is the primary focus of this proposal. The compiler also produces a host C program, which is used to drive the entire system. In order to produce an executable, the VHDL is processed by a set of commercial tools – the Synplify synthesis compiler and the Xilinx M1 place and route tools - to produce the reconfigurable binary file, and the host code is processed by a standard `gcc` compiler.

[1]Several people have contributed to the work described in this chapter. Margaret Carter wrote the initial implementation of the DFG to VHDL translator. Monica Chawathe has added new functionality to the translator, including the addition of many new nodes, the "glue code" generation, and the Synplify project file generation. Amit Patel wrote the `Collect` component. Charlie Ross wrote much of the memory access code, including the `MemArb` and `ConstGrabber` components. None of the project would be possible without the excellent SA-C compiler, written primarily by Jeff Hammes, and the run-time system and DFG simulator, written by Harish Kantemneni and Dave McClure.

Finally, the compiler can optionally produce a host-only executable, called a C-dump, of the SA-C program – this can be used for program debugging and verification.

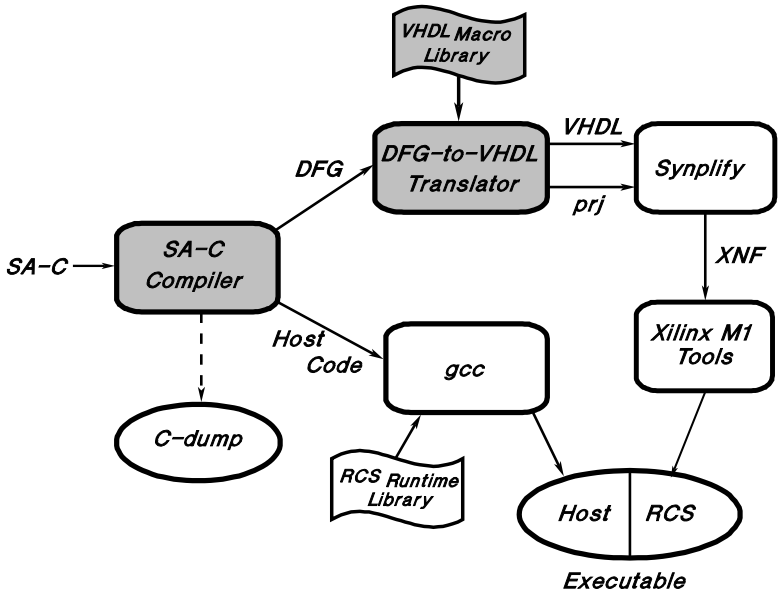


Figure 4.1: Overview of the SA-C Compilation process

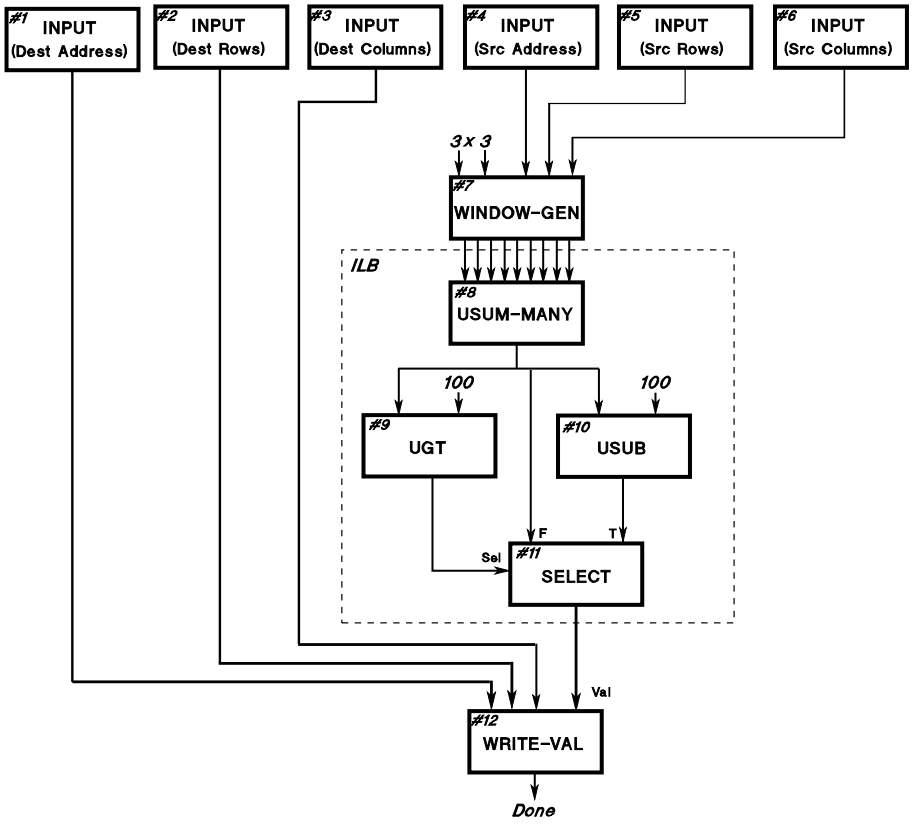
4.1 The SA-C Compiler and Optimizations

While the focus of this proposal is on the DFG-to-VHDL translation, a little more information about the compiler, the dataflow graphs it creates, and the optimizations it performs are necessary. A simple SA-C program is shown in Figure 4.2(a). This program accepts a 2-D array (named `Arr`) of 8-bit unsigned integers (i.e., of type `uint8`) as input. A window generator statement (`for window...`) extracts all 3×3 sub-arrays from the image array, sums the elements in each sub-array. A new array (named `r`) is formed whose elements are either the sum of the corresponding window, or if the sum is greater than 100, the value is sum minus 100. This rather nonsensical program demonstrates several characteristics of the SA-C language; a complete language reference is available in (Hammes and Böhm 1999). Upon compilation, a DFG of the program is generated; a pictorial representation of the DFG for this program is shown in Figure 4.2(b).

Numerous optimizations are performed by the compiler before creating the DFG; many have been motivated by the experience gained from the manually coded Prewitt project

```
uint8[:,:] main (uint8 Arr[:,:]) {
 uint8 r[:,:] =
   for window W[3,3] in Arr {
     uint8 s = array_sum (W);
     uint8 v = if(s>100) return(s-100)
               else      return(s);
    } return (array (v));
  } return (r);
```

(a)



(b)

Figure 4.2: (a) A simple SA-C program, and (b) the resulting dataflow graph

discussed earlier. Several traditional optimizations help to minimize the amount of calculation required by the hardware, including code motion, constant folding, array and constant value propagation, and common subexpression elimination. Function inlining and loop unrolling provide parallel computation opportunities. Other, less traditional optimizations are included to improve hardware size and speed performance:

- Array and loop size propagation, facilitated by the use of the loop generator statements, help to insure that any hardware registers that must be synthesized are of minimum width.

- Similarly, bit width narrowing works in conjunction with loop unrolling to insure that each instance of a loop body uses the smallest data sizes possible to perform calculations.

- *Stripmining* - splits a loop into a pair of nested loops, with the outer loop creating chunks of work which are carried out by the inner loop. When implemented in hardware, parallelism is introduced when the inner loop bodies are instantiated individually, and the outer loop is converted into code which distributes data to these instances. This technique also allows a large image to be split into smaller pieces that fit into the RCS memory.

- *Tiling* - allows several iterations of a loop to be grouped together. Then, like stripmining, each iteration of the group can be instantiated separately to increase parallelism. When combined with stripmining, tiling allows the compiler to produce code which effectively utilizes the entire bandwidth of the target hardware. This provides the automated mechanism which is analogous to the grouping of 2 (or 6) Prewitt calculations in the manual Prewitt project to better utilize the hardware.

- Loop fusion allows two or more consecutive loops in an algorithm to be automatically joined together. This helps to eliminate extra data transmissions between processing steps. The compiler can often perform this operation in situations where such fusion is not obvious to the application programmer.

Some of the optimizations may be detrimental to performance if applied in the wrong situation, or trade one hardware resource for another; in these cases, their application can be controlled by *pragmas*.

Once these optimizations are performed, their effects are reflected in the resulting DFG. The DFG to VHDL translator has no direct knowledge that these optimizations have taken place.

4.2 An Abstract Architecture for Reconfigurable Computing

Unlike "standard" instruction set architectures, which provide a relatively small set of well-defined instructions to the user, RCS's are composed of an amorphous mass of logic cells which can be interconnected in any number of ways. To limit the number of possibilities available to the designer, an *abstract architecture* has been defined as a more reasonable target. This abstract architecture also provides some degree of hardware independence, which will facilitate the transfer of a design to different hardware platforms.

The abstract architecture defines three main categories of functions:

- Data transfer mechanisms. These include streams, block memory transfer, and systolic. These can be applied both between host and RCS as well as within sections inside the RCS.

- Arithmetic and Logical Operations. These include common simple operators such as `ADD`, `SHIFT`, and `COMPare`, as well as more complex operations such as `SQRT`, `SUM`, and `MEDIAN`. The set of complex operations that were selected was influenced by the target application domain – image processing.

- Data buffering and storage mechanisms, including FIFOs, and arrays of buffers, which can be interconnected in various ways, such as shift registers.

The functions comprising the abstract architecture that were deemed necessary for the implementation of the compilation system were selected first; these functions helped to determine those parts of the real hardware that were used in the initial implementation.

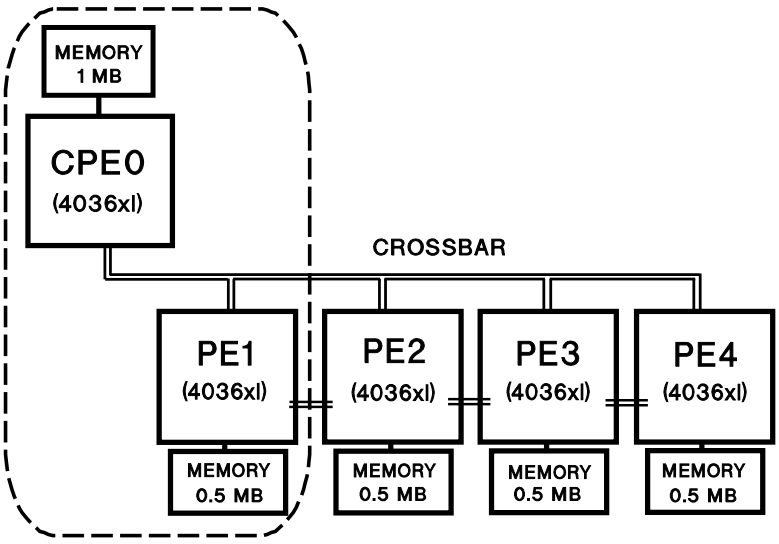


Figure 4.3: Architecture of the Wildforce-XL Reconfigurable Computing Board, showing that part of the board being utilized by the DFG-to-VHDL translator

4.3 Implementation on the AMS Wildforce Board

The first version of the compilation system targets the AMS Wildforce board. By virtue of the experience derived from the manual Prewitt project, a subset of the resources available on the board have been chosen for the implementation. Figure 4.3 shows the board used as a target – the first implementation of an automated system uses only that portion of the board shown inside the dotted lines in the figure. This implementation model was chosen primarily for simplicity – CPE0 retrieves image data from its local memory and sends it in the proper order over the crossbar to PE1, which buffers it (using the shift register scheme described above), presents it to the top of the ILB, and stores the results in its local memory. Since two PEs, and thus two memories (one for reading and one for writing) are used, the design is simplified because there is no memory contention between reading and writing. It is expected that the next generation of the translator will use more of the PEs on this board, either for added functionality or to enhance parallelism in the present system. On the other hand, this subset seems reasonable in its own right, since the trend in new board designs is to use fewer, albeit larger, FPGAs; therefore, this scheme seems to be a reasonable choice for future work when the system is ported to other hardware.

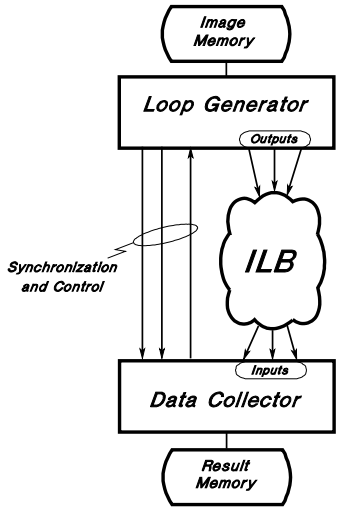


Figure 4.4: Structure of the VHDL code generated by the DFG-to-VHDL translator

The choice of using PE1 in the implementation is arbitrary here – any of the PEs 1–4 could have been used, since they are identical chips and have nearly identical connections. For this reason, the following discussion uses the term *PEx* to reference this PE.

4.4 Translating DFGs to VHDL

At first glance it appears that the DFG of Figure 4.2(b) describes a simple top to bottom execution that can be implemented by a combinational circuit; however, the operation of some of the nodes are more complicated. In particular, the `WINDOW-GEN` node must retrieve data from RCS memory and present it in the proper order to the ILB of the program, and the `WRITE-VAL` node must collect the results and store them in the proper order into memory so the host can retrieve them. Each of these operations require multiple clock cycles, several state machines, and coordination between the upper (loop generator) and lower (data collection) nodes. Figure 4.4 shows a conceptual picture which illustrates the resulting design partitioning.

4.5 Classification of DFG Nodes

As a first step in converting the DFG to VHDL, the translator classifies the dataflow graph nodes into four types:

- Run-time input nodes. These nodes are not directly translated into VHDL, but rather specify addresses of memory locations within CPE0's memory where run-time data can be found. These addresses are allocated by the compiler, and include information whose exact values are only known at run-time, such as the size, shape, and starting address of the data arrays. The translator uses this information to form a table of addresses which will be used by the `ConstGrabber` module to retrieve the data from memory during execution.

- Generator nodes. The information in these nodes help the translator to choose the VHDL generator components which are appropriate for the particular design. In contrast to the run-time constants discussed above, which are retrieved and used during execution, the information provided by these nodes is used during the VHDL compilation, and are used to parameterize the instantiation of the various VHDL components. This information includes such things as window size, shape, and step size.

- Loop body nodes. These nodes specify the operations to be performed by the ILB. These nodes are used to generate the VHDL code for the ILB.

- Reduction nodes. Similar to the generator nodes, these specify the parameters needed to select and instantiate the reduction (collection) nodes.

Given this information, the translation process is divided into three main parts. First, the ILB is identified as being that part of the DFG that lies between the *outputs* of the loop generator nodes and the *inputs* of the data collection nodes, and consist entirely of loop body nodes. This section of the DFG is translated directly into a VHDL *component*. Then, the loop generator and collection nodes are implemented by selecting the proper VHDL components from a library, and by supplying these components with a parameters file containing information derived from the pertinent DFG nodes. Finally, the translator specifies the interconnections between the ILB and generator/collection components by generating two top–level VHDL modules, one for CPE0 and one for PEx, and a set of *project* files used by the Synplify VHDL compiler and synthesis tool. The files created in

```
entity Computation is
  port(NODE7OUT : in ByteArr(WinSize-1 downto 0);      -- Line 1
       NODE12OUT: out Byte);                           -- Line 2
end;                                                   -- Line 3
                                                       -- Line 4
architecture byblocks of Computation is                -- Line 5
  signal USUM_MANY8OUT0: std_logic_vector(7 downto 0); -- Line 6
  signal UGT9OUT0: std_logic;                          -- Line 7
  signal USUB10OUT0: std_logic_vector(7 downto 0);     -- Line 8
  signal SELECTOR11OUT0: std_logic_vector(7 downto 0); -- Line 9
begin                                                  -- Line 10
  NODE9: USUMMANY generic map( nVals => 9,             -- Line 11
                               insize => 8,            -- Line 12
                               outsize=> 8)            -- Line 13
                 port map( vals => NODE7OUT,           -- Line 14
                           result => USUM_MANY8OUT0);  -- Line 15
  UGT9OUT0 <= '1' when USUM_MANY8OUT0 > 100            -- Line 16
                  else '0';                            -- Line 17
  USUB10OUT0 <= USUM_MANY8OUT0 - 100;                  -- Line 18
  SELECTOR11OUT0 <= USUB10OUT0 when UGT9OUT0 = '1'     -- Line 19
                    else USUM_MANY8OUT0;               -- Line 20
  NODE12OUT<= SXT(SELECTOR11OUT0, 8);                  -- Line 21
end;                                                   -- Line 22
```

Figure 4.5: The generated VHDL for the ILB of Figure 4.2(b) (some declarations and type casting have been omitted for clarity).

this last step serve to "glue" all the components together into a final design.

4.6 Translation of the ILB

The translation of the ILB involves a traversal of the dataflow graph. A VHDL *component* is created whose inputs are connected to the outputs of the loop generator node, and whose outputs are connected to the inputs to the data collection node. As an example, Figure 4.5 shows the VHDL that is generated for the ILB denoted by the dotted lines in the DFG of Figure 4.2(b).

Many nodes implement simple operations, such as addition or logical operations; for these nodes, there is a one-to-one correspondence between DFG node and VHDL statement; line 18 in Figure 4.2 is an example – the subtract is handled in a single line. For more complicated operations, such as the SA-C reduction operations like *array sum*, the translator generates a connection to a VHDL component; see lines 11-15. A library of such

components has been written directly in VHDL; this allows a SA-C program access to operators that either cannot be expressed in the high level language or that have efficient direct hardware implementations. To facilitate the tracing of signals through the ILB, the names of the signals used to interconnect nodes are derived from the DFG node type and number. For example, the `signal` name `UGT9OUT0` in the VHDL example corresponds to the first (number 0) output of the `UGT` node numbered 9 in the DFG.

At present, the generated ILB is entirely combinational, although it is expected to eventually include multiple-cycle functions such as lookup tables, complex data reduction operations, and pipeline registers.

4.7 Implementation of the Other Components in a Design

In contrast to the ILB, which is generated from scratch by traversing the DFG, the data generators and collectors are composed of pre-built VHDL components which are selected by the translator from a module library, and are parameterized with values from the DFG. An entire implementation, including the top-level glue modules mentioned earlier, is shown pictorially in Figure 4.6. The operation of each component in this figure will be discussed in the following sections. The general flow of information is from top left to bottom right – the following discussion follows roughly the same order.

4.7.1 The `ConstGrabber` Component

Prior to initiating execution of the hardware, the host C program, which is created by the compiler along with the DFG, is responsible for transferring the run-time constants and input data to CPE0's memory, and then resetting the hardware to its initial state. After the system is reset, the `ConstGrabber` component reads the run-time constants from memory, and makes them available to the rest of the system. The memory used by these constants is allocated by the compiler, and the addresses are communicated to the translator by the DFG `INPUT` nodes. Once all the constants have been retrieved, the module sends a `Ready` signal to the rest of the system.

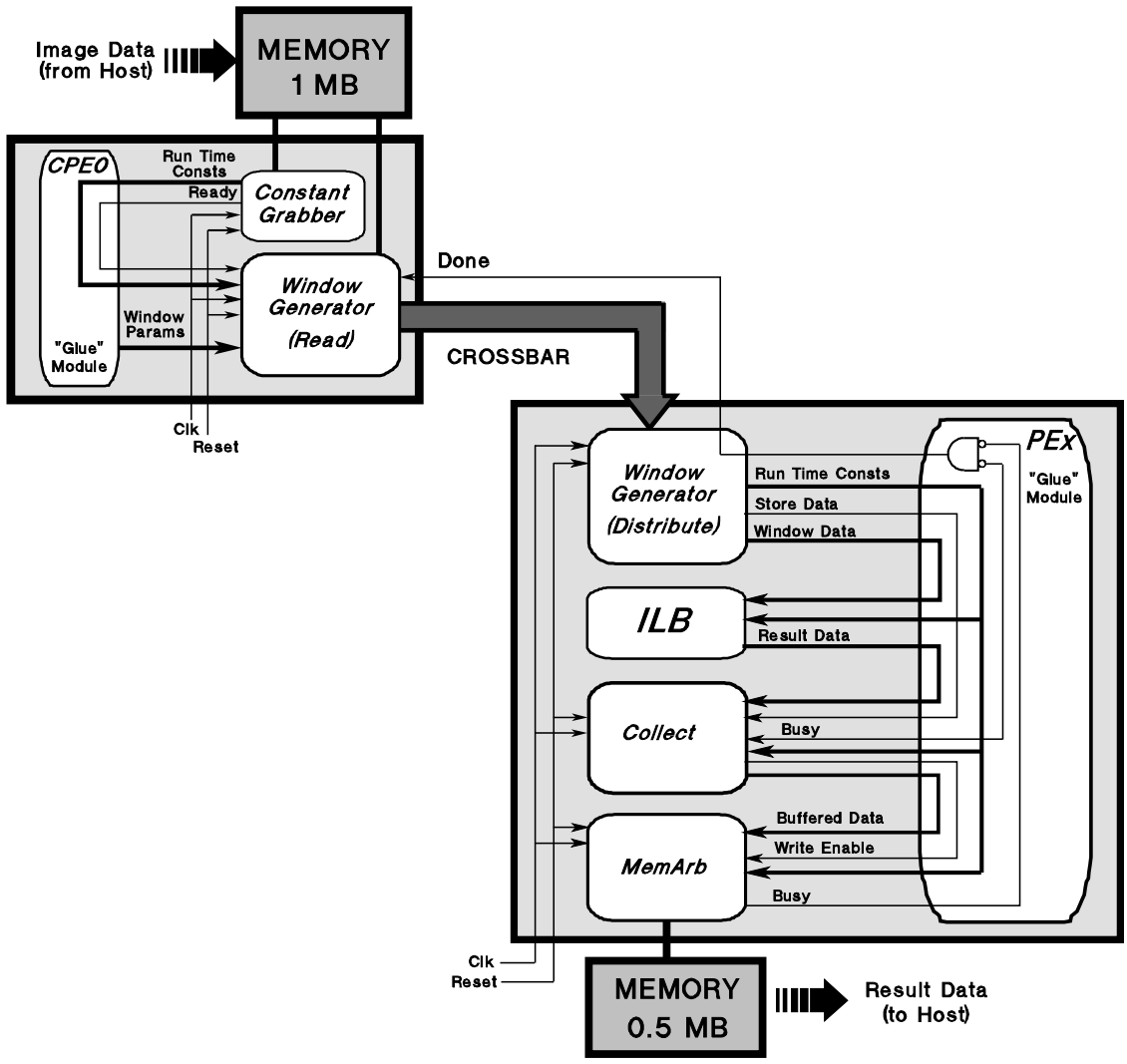


Figure 4.6: Location of components in the 2 PE model

4.7.2 The Data Generator Components on CPE0

The data generator is responsible for retrieving data from memory in the proper order and buffering it, so it can serve as input to the inner loop body (ILB). There are three types of data generators – scalar generators, which generate values similar to the traditional `for` loops found in C and other languages; element generators, which extract single values from an array, and window generators, which produce small sub-arrays extracted from a larger array. The discussion that follows specifically describes the window generator; however, the other generators operate in a similar fashion (in fact, the element generator is currently implemented as a window generator with a 1×1 window).

The window generator function is distributed over the two PEs. The part that resides on CPE0 (called the *read* function) is the more complicated of the two parts; it is responsible for reading data from CPE0's memory and placing the data on the crossbar. It consists of two separate state machines – one which is responsible for retrieving data from memory (the *ReadData* state machine), and a second that sends the retrieved data out to the crossbar (the *XBar* state machine).

Initially, the `XBar` state machine waits for the `Ready` signal from the `ConstGrabber` module, then sends the retrieved run-time constants on the crossbar, so they can be stored and used by the components in PEx. Once this initial step occurs, the main operation of the read function starts. Figure 4.7 illustrates this operation for a 3×3 window example.

The `ReadData` state machine begins reading words of data from memory. The number of words read is equal to the number of columns in the window being used, and as called a "frame" of data. As the data is being read, it is stored in a temporary buffer (called `TmpBuf`); once all the words for a complete frame have been read, they are transferred to a second buffer called `InBuff`. This double buffering allows one frame to reside within CPE0 while the next frame is being retrieved from memory. Once a complete frame has been input, `ReadData` sends a signal to the `XBar` state machine to indicate that the frame is ready to be sent to PEx. `ReadData` then initiates the reading of the next frame of data.

Upon receiving the signal from `ReadData`, the `XBar` machine starts sending data along the crossbar. Data is stored in memory and therefore retrieved in row-major order; how-

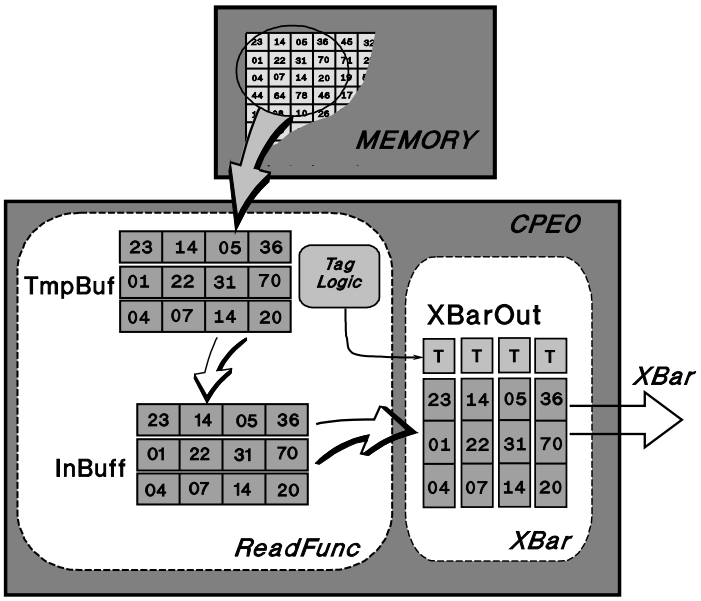


Figure 4.7: Diagram of the read window generator function

ever, it is sent along the crossbar in columns. An interconnection network, generated from the static parameters passed to the component at compilation, performs the transposition from rows to columns. The resulting buffer array is named `XbarOut`, and holds the data that is output to the crossbar.

Several signals are generated and sent as tag bits as `XBar` sends the data; these signals are used by PEx to help interpret the data:

- `ValidData` (crossbar bit 35) - Indicates that the value on the crossbar is a legal data item. When set to 0, the value currently on the crossbar is undefined (i.e. a `NULL`). `NULL` are sent during cycles when no valid data is available to be sent, such as when `XBar` is waiting for data from `ReadData`.

- `Start/Stop` (bit 34) - Set to indicate the beginning and end of data.

- `DontStore` (bit 33) - Used to indicate that the calculations calculated by the ILB at this time step should not be stored. This situation occurs at the beginning of each row, and when the window generator is using a step size other than 1, as described below.

- `LastCol` (bit 32) - set to one when the values being sent are from the last column

in a row.

In addition to sending window data in the proper sequence, the window generator code on CPE0 must handle several special situations. Since the rate at which the generator produces values determines the overall data rate of the system, it is possible for windows to be created faster than the result data can be stored. This might occur when an ILB produces several output values for each input. In this case, `XBar` places `NULL` values on the crossbar to "slow down" the window generation rate. To implement this, the number of cycles required to do all the writes is calculated by the translator and passed as a generic to the component; this value is used to determine the number of `NULL`s (if any) that must be set after each frame to keep the data generation rate from overrunning the output bandwidth.

The system must also handle window steps other than 1 – for example, a step of 2 specifies that only every other window is supposed to produce a value. In these situations, all of the data is still sent, and the ILB, being a combinational circuit, still calculates values at each time step; however, only some of the calculated results should be stored. The `DontStore` signal is set to a `1` for each cycle when the value should not be stored. `DontStore` is also used to prevent the storing of results calculated at the beginning of each row, before the first complete window of data has been transmitted along the crossbar.

4.7.3 The Data Generator Components on PEx

That portion of the data generator that resides on PEx is called the *distribute* function. It retrieves the data from the crossbar and buffers it, so it can be presented to the inputs of the ILB. It also uses the tag bits sent on the crossbar to generate signals which control the other components on PEx. The distribute function consists of a single state machine.

In concert with operation of the code on CPE0, the `Distribute` state machine retrieves the run-time data from the crossbar, and stores the retrieved values into an array; the translator uses information it gleans from the DFG to create the PEx glue code; this code "wires" the various values in the array to the appropriate inputs of the other PEx components. Once this step is finished, `Distribute` goes into its "normal" operation.

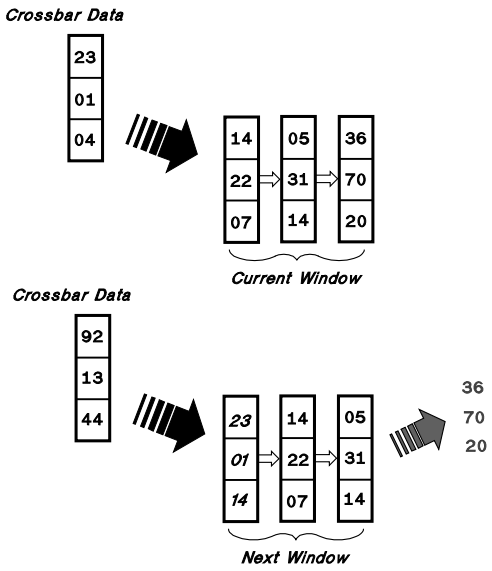


Figure 4.8: Pictorial representation of a 3×3 sliding window generator for the current, and then next windows.

The heart of the `Distribute` function is a shift register which buffers the window data. its operation is illustrated in Figure 4.8. A "sliding window" effect is created by the shift register – when new data arrives from the crossbar, a new window is formed by taking the previous columns that had been sent, shifting them over, and then placing the new column of data into the space just vacated by the shift operation.

`Distribute` also generates a signal called `Storedata`, which is derived from the `ValidData and DontStore` tags bits from the crossbar. This signal inhibits the storing of data at the beginning of each row and during window stepping.

One special operator used with data generators is the `dot` (or dot product, although the operator does not imply that a product or any other arithmetic operation is to be performed). Consider the following SA-C statement:

```
for elem1 in image1 dot elem2 in image2
    .
    .
```

This dot operator syntax specifies that two (or more) data generators are to operate simultaneously, thereby providing two (or more) sets of data to the ILB at the same time. The latest version of the generator code has extended the window generator code to handle

certain combinations of the `dot` operation. The operation of this generator is very similar to that of the single window. The primary difference is that a separate set of buffers on both CPE0 and PEx are instantiated for each generator. As the `XBar` machine sends data on the crossbar, first a column from one window is sent, then a column from the next window is sent, etc. The `distribute` machine on PEx then places the data retrieved from the crossbar into the proper shifter register.

4.7.4 The `Collector` Component

Once data has been presented to the input of the ILB, the resulting values must be stored away into memory. The `Collector` component consists of a state machine which performs this function. The `StoreData` signal generated by `Distribute` causes the result value to be placed into a buffer; once an entire word of data has been collected, the buffer is sent to the `MemArb` component, along with a write enable signal.

In addition to single values, the *collector* can also handle *tiles* of results. Tiles are small arrays of data, similar to windows, that can be produced by an ILB. Usually tiled output occurs as a result of the stripmining and tile optimizations applied by the compiler – an original (single) ILB is transformed into several instantiations, with the resulting output being a tile whose size and shape is determined by the order in which the compiler optimizations were applied. The tile shape is determined at compile time, so buffers are instantiated within `Collector` to hold the resulting tile values.

A different situation occurs for a single ILB which produces multiple (independent) values. In this case, a separate `Collector` component is instantiated for each output. This method is used so that the proper size and shape of buffers can be independently instantiated for each output. The PEx glue code is responsible for providing the interconnection for each instance of the `Collector` component.

With multiple and tiled outputs being produced by the ILB, it is possible that a single set of input values can produce multiple outputs, thus causing the output bandwidth to exceed the input. The `Collector` function does not need to worry about this, since the window generator functions will add wait states if necessary to guarantee that the input

rate will not cause data overrun on output.

4.7.5 The `MemArb` Component

The final component in the system is the `MemArb` component. This module receives words of data to be stored in memory, and performs the storing operation. In contrast to the `Collector`, which may have several instantiations for multiple outputs, there is always only one `MemArb` component; however, it must handle multiple store requests from possibly several `Collectors`. A simple priority scheme is used to determine who goes first – the output request which is wired to the highest priority input to the `MemArb` component goes first. Since the window generators never produce data faster than the results can be stored, starvation of the lowest priority inputs is avoided.

4.8 An Automated Prewitt Implementation on the AMS Wildforce Board

Since the Prewitt algorithm was discussed earlier in the manual project, it is interesting to compare an automated implementation with the manual one. A SA-C program which performs the Prewitt calculation is shown in Figure 4.9.

The SA-C compiler performs several of the optimizations described earlier on this program before producing the DFG. Since the convolutions involve multiplications with 3×3 masks that are composed of the constants 1, 0, and -1, the compiler optimizes the calculation to a series of additions and subtractions. Multiplications with zero are eliminated completely. These optimizations eliminate all multiplies, and reduce the number of addition/subtractions from 16 down to 10, very similar to the manual optimizations performed in the previous project.

The `magnitude` function is the most expensive part of the ILB, since it involves the squaring of the two results (requiring a multiply), then finding the square root. Unlike the manual method, the system is not able to produce lookup table implementations, so the magnitude calculations must be computed the hard way. An efficient square root routine is used which uses only shifts, adds, and bit operations. Nonetheless, the multiply/square-root operation consumes around 75% of the time required by the entire ILB.

```
uint8[:,:] prewitt(uint8 Image[:,:]) {
           int2 H[3,3] = {{-1,-1,-1},
                          { 0, 0, 0},
                          { 1, 1, 1}};
           int2 V[3,3] = {{-1, 0, 1},
                          {-1, 0, 1},
                          {-1, 0, 1}};
 uint8 res[:,:] =
   for window W[3,3] in Image {
     int11 sh, int11 sv =
      for h in H dot w in W dot v in V
        return (sum( (int11)w*h ),
                sum( (int11)w*v ));
    } return (array (magnitude(sh,sv)/8));
  } return (res);

uint11 magnitude(int11 a, int11 b)
  return (sqrt( (int22)a*a + (int22)b*b ));
```

Figure 4.9: The Prewitt edge-detection algorithm, written in SA-C

The resulting DFG is processed by the DFG-to-VHDL translator, which extracts the ILB and translates it directly to VHDL, selects the appropriate generator and collector components from the VHDL library, and creates the top-level VHDL program which "glues" the entire system together. The translator also creates the script files needed by commercial design tools to compile and place-and-route the VHDL into FPGA configuration codes. These files, along with a compilation script that controls the numerous steps in the compilation process, allows the entire process, from high level language compilation down to the production of FPGA configuration codes and the host-based control program, to be fully automated. The user can execute the entire algorithm on the hardware like any other application (by typing `a.out` or something similar), without needing to worry about any of the operational details of the hardware.

Table 4.1 shows some of the statistics of the entire compilation/translation process. The 19 line SA-C program for the Prewitt algorithm eventually requires over 5000 lines of VHDL, and occupies approximately 30% of the CLBs in the two FPGAs used in the implementation.

The entire design results in an effective processing rate of nearly 1 MegaPixels/sec.

Lines of Code

SA-C	19
VHDL	
`WINDOW-GEN` - PE0	572
`WINDOW-GEN` - PE1	194
Generated Inner Loop Body	3744
`WRITE-VAL`	406
Glue Code and Misc	199
Total VHDL	5115

CLB Usage

`WINDOW-GEN` - PE0	251	(19.4%)
`WINDOW-GEN/WRITE-VAL` - PE1	236	(18.2%)
Inner Loop Body	281	(21.7%)
Glue Code and Misc	19	(1.5%)
Total - PE1	536	(41.3%)
Total CLBs	787	(30.4%)

Propagation Delay - Inner Loop Body

`ILB - Convolution`	38.1 ns	
`ILB - Magnitude`	165.9 ns	
Total Propagation Delay	214.0 ns	(4.7 MHz)

Execution Time on Wildforce Hardware

Image Size	240×200	272×400
Image Download	0.027 sec	0.035 sec
Computation	0.024 sec	0.054 sec
Result Upload	0.009 sec	0.021 sec
Total time	0.060 sec	0.110 sec
Computation Rate		
Computation Only	2.4 MPixels/s	2.01 MPixels/s
Effective (w/ I/O)	0.80 MPixels/s	0.99 MPixels/s

Table 4.1: Statistics for the Wildforce implementation of the Prewitt algorithm using the SA-C compiler/translator.

While this doesn't appear to be very close to the more than 5 MegaPixels/sec rate achieved by the manual implementation, it does not include any parallelism (tiling or stripmining) optimizations in the design, and does not use lookup tables or pipelining, which were important optimizations used in the manual implementation. It seems reasonable to expect

that the performance can be improved to the point that the automated process can produce results that compete favorably with the manual design.

Another very important observation should be made concerning the manual vs. automated approach. The entire Prewitt algorithm was coded in SA-C and converted to hardware in a matter of minutes or at most hours, compared with at least two months required by the manual method. Equally important, the development process can be carried out by an application programmer, with little or no knowledge of hardware design or VHDL. Clearly, the automated process can help to revolutionize the development process for reconfigurable hardware.

4.9 Performance of the Level 1 Benchmark Routines

As a final discussion of the current compilation/translation system, some results from compiling some typical image processing applications will be discussed. A set of "typical" IP operations were written by the Image Processing group within the Cameron project. These programs ranged from very simple operations (such as adding two images together to form a new image) to moderately complex (the Prewitt, Roberts and Sobel algorithms for edge detection, various filter operations). They were executed on the Wildforce board, as well as stand-alone on the host computer (a 266MHz Pentium).

This set of benchmark programs require a significant subset of the SA-C language to be compiled and translated – its first use was to verify that the compiler/translator could actually compile the codes. Work is continuing in analyzing the results.

Table 4.2 summarizes the amount of FPGA space (in terms of CLBs) required by several of the benchmark programs. Table 4.4 show the execution time for each benchmark, along with the execution time of the same program when compiled for host-only execution. Some preliminary conclusions can be drawn from the results:

- Even though the benchmark codes are relatively simple, they provided a fair challenge to compile and translate. However, as a result of this exercise, the entire system has become much more robust.

- Most of the codes fit onto the existing Wildforce hardware. The ones that did not fit required a significant amount of data buffering, which has already identified as being expensive. Some codes would only fit by rewriting them, or by turning off some of the optimizations, such as loop fusion, that promise more performance but that do require more buffer space. This situation may be mitigated by new compiler functionality being considered, as well as by some of the optimizations proposed here. New technology promises to help alleviate this problem, as it provides an order of magnitude more space than is currently available.

- In terms of execution speed, results were mixed. For the simple operations, such as AddDyadic (add two image arrays together), the transmission time of the image arrays to and from the Wildforce board is a significant fraction of total execution. As a result, these benchmarks run as much as twice as slow overall as the host-only versions, even though the actual computation time was several times faster. For more complicated (and more realistic) codes, the hardware ran up to 12 times faster than the host-only code, even after the array transmission time was amortized into the times. This suggests that RCS may be useful in certain applications.

This results reported here are preliminary, and reflect a "vanilla" compilation for the codes. The performance numbers can serve as a baseline for measuring the effectiveness of new functionality and optimizations for the system. It is expected that applying the more aggressive optimizations, such as tiling and stripmining, that are designed to better utilize the bandwidth of the hardware, will produce significantly better results.

	Benchmark Name	Space CPE0			Space PEx			Freq (MHz)
1	AddDyadicI8O8	F 775 H 153 FF 639		30%	F 564 H 45 FF 186		22%	10.5
2	AddMonadicI8O	F 525 H 128 FF 461		20%	F 355 H 44 FF 186		14%	9.99
3	CloseI8O8W3x3	F 1126 H 153 FF 1289		50%	F 2863 H 51 FF 499		111%	
4	ConvolutionI8O20W3x3	F 850 H 238 FF 872		34%	F 1565 H 21 FF 274		60%	9.98
5	ConvolutionI8O21W5x5	F 1281 H 452 FF 1515		58%	F 3577 H 41 FF 570		138%	
6	ConvolutionI8O22W7x7	F 1897 H 784 FF 2413		93%	F 6556 H 46 FF 925		253%	
7	DilationI8O8W3x3	F 1056 H 329 FF 1161		45%	F 792 H 68 FF 307		31%	10.30
8	ErosionI8O8W3x3	F 1056 H 329 FF 1161		45%	F 797 H 80 FF 307		31%	9.79
9	GaussianFilterI8O12.4W3x3	F 628 H 136 FF 579		24%	F 555 H 33 FF 218		21%	9.94
10	LaPlaceFilterI8O12W3x3	F 628 H 136 FF 579		24%	F 644 H 46 FF 218		25%	10.39
11	LaPlaceFilterI8O15W5x5	F 699 H 170 FF 715		28%	F 1320 H 48 FF 386		51%	10.11
13	MaxFilterI8O8W4x5	F 715 H 143 FF 643		28%	F 738 H 56 FF 314		29%	10.45
15	MedianFilterI8O8W5x5	F 697 H 172 FF 707		27%	F 4374 H 36 FF 394		169%	

Table 4.2: Space required by several of the Level 1 benchmarks. The table shows the number of F-Maps, H-Maps, and register (Flip/Flop) bits required for CPE0 and PEx for each routine, along with the percentage of CLBs within each PE that is used. Also, the operating frequency for each program is shown.

	Benchmark Name	Space CPE0			Space PEx			Freq (MHz)
16	MinFilterI8O8W3x4	F H FF	627 133 579	24%	F H FF	660 53 250	26%	9.48
17	MinI8O8	F H FF	473 129 411	19%	F H FF	61 8 50	3%	13.37
18	MultDyadicI8O16	F H FF	775 153 639	30%	F H FF	594 35 178	23%	10.38
19	MultMonadicI8O16	F H FF	525 128 461	20%	F H FF	584 30 162	23%	10.09
20	OpenI8O8W3x3	F H FF	1126 384 1289	50%	F H FF	2592 39 499	111%	
21	PrewittMagI8O8	F H FF	630 140 579	24%	F H FF	1488 76 245	57%	2.41
24	RobertsMagI8O8	F H FF	583 136 515	23%	F H FF	1118 66 198	43%	2.98
25	SobelMagI8O8	F H FF	630 140 579	24%	F H FF	1605 72 233	62%	2.25
28	SubDyadicI8O9	F H FF	775 153 639	30%	F H FF	499 35 178	19%	10.40
29	SubMonadicI8O8	F H FF	525 128 461	20%	F H FF	487 32 162	19%	10.26

Table 4.3: Continuation of Table 4.2

	Benchmark Name		Caboose 300 × 198	Camlogo 461 × 221	Towngrey 665 × 699
		Write	0.0131	0.0224	0.1018
		Compute	0.0113	0.0195	0.0887
1	AddDyadicI8O8	Read	0.0113	0.0195	0.0880
		Total	0.0357	0.0614	0.2785
		SC-code	0.0208	0.0339	0.1643
		Write	0.0066	0.0113	0.0509
		Compute	0.0063	0.0108	0.0494
2	AddMonadicI8O	Read	0.0110	0.0195	0.0877
		Total	0.0238	0.0416	0.1880
		SC-code	0.0163	0.0280	0.1312
		Write	0.0066		
		Compute	0.0115		
4	ConvolutionI8O20W3x3	Read	0.0113		
		Total	0.0440		
		SC-code	0.1319		
		Write	0.0066	0.0113	0.0509
		Compute	0.0058	0.0099	0.0452
7	DilationI8O8W3x3	Read	0.0112	0.0191	0.0873
		Total	0.0236	0.0403	0.1838
		SC-code	0.1873	0.3119	1.4862
		Write	0.0065	0.0113	0.0509
		Compute	0.0060	0.0103	0.0476
8	ErosionI8O8W3x3	Read	0.0112	0.0192	0.0870
		Total	0.0237	0.0408	0.1857
		SC-code	0.1875	0.3195	1.485
		Write	0.0066	0.0113	
		Compute	0.0119	0.0206	
9	GaussianFilterI8O12.4W3x3	Read	0.0221	0.0379	
		Total	0.0406	0.0697	
		SC-code	0.3868	0.7277	
		Write	0.0066	0.0113	
		Compute	0.0114	0.0196	
10	LaPlaceFilterI8O12W3x3	Read	0.0221	0.0379	
		Total	0.0401	0.0688	
		SC-code	0.2065	0.3547	

Table 4.4: Execution times of several of the Level 1 benchmarks. The table shows separate times for the array write to the Wildforce local memory, the computation, and the reading of result data back to the host, for three different-sized images. For comparison purposes, the execution time of the same program on the host computer is also shown.

	Benchmark Name		Caboose 300 × 198	Camlogo 461 × 221	Towngrey 665 × 699
		Write	0.0066	0.0113	
		Compute	0.0116	0.0200	
11	LaPlaceFilterI8O15W5x5	Read	0.0217	0.0375	
		Total	0.0399	0.0688	
		SC-code	0.5442	0.9359	
		Write	0.0066	0.0113	0.0509
		Compute	0.0113	0.0194	0.0891
13	MaxFilterI8O8W4x5	Read	0.0110	0.0190	0.0890
		Total	0.0289	0.0497	0.2290
		SC-code	0.1554	0.2400	1.2384
		Write	0.0066	0.0113	0.0509
		Compute	0.0063	0.0107	0.0492
16	MinFilterI8O8W3x4	Read	0.0112	0.0191	0.0874
		Total	0.0241	0.0411	0.1876
		SC-code	0.1039	0.1497	0.8203
		Write	0.0066	0.0113	0.0509
		Compute	0.0045	0.0077	0.0350
17	MinI8O8	Read	0.0000	0.0000	0.0000
		Total	0.0111	0.0190	0.0860
		SC-code	0.0050	0.0091	0.0423
		Write	0.0131	0.0226	
		Compute	0.0115	0.0198	
18	MultDyadicI8O16	Read	0.0232	0.0385	
		Total	0.0478	0.0809	
		SC-code	0.0189	0.0319	

Table 4.5: Continuation of Table 4.4

	Benchmark Name		Caboose 300 × 198	Camlogo 461 × 221	Towngrey 665 × 699
19	MultMonadicI8O16	Write	0.0066	0.0113	
		Compute	0.0059	0.0102	
		Read	0.0224	0.0385	
		Total	0.0349	0.0600	
		SC-code	0.0155	0.0267	
21	PrewittMagI8O8	Write	0.0066	0.0113	0.0509
		Compute	0.0245	0.0423	0.1936
		Read	0.0116	0.0193	0.0869
		Total	0.0427	0.0729	0.3315
		SC-code	0.4308	0.9071	3.407
24	RobertsMagI8O8	Write	0.0066	0.0113	0.0510
		Compute	0.0199	0.0343	0.1584
		Read	0.0113	0.0193	0.0872
		Total	0.0378	0.0649	0.2966
		SC-code	0.2613	0.3927	2.0201
25	SobelMagI8O8	Write	0.0066	0.0114	0.0510
		Compute	0.0262	0.0451	0.2066
		Read	0.0117	0.0192	0.0870
		Total	0.0445	0.0758	0.3446
		SC-code	0.4305	0.7044	3.4015

Table 4.6: Continuation of Table 4.4

Chapter 5

Pipelining the Inner Loop Body

The automated system described in the last chapter is functional, and produces reasonable hardware "designs" for many applications. This chapter describes an optimization to the basic functionality of the system – pipelining the inner loop body. In contrast to the numerous optimizations that are performed by the compiler before it generates the DFG, this optimization is performed on the dataflow graph itself. This optimization is appropriately applied here, since, unlike the compiler optimizations that use semantic information to perform its optimizations, and are therefore independent of any particular hardware, this optimization relies on timing information from the RCS system itself.

5.1 Introduction

One characteristic that has been observed in inner loop bodies, as currently produced by the compiler, is the tendency to have long propagation delays, at least for non-trivial problems. This delay determines the overall operating frequency of the resulting design, so the design runs slowly. Worse yet, some designs require a clock rate of less than 2 MHz. This is slower than the clock generator is capable of running on the present hardware, meaning that the design no longer functions properly. The problem can sometimes be mitigated by recoding the application to use simpler operations, or by breaking the ILB into smaller pieces. However, it is not always obvious, at the source code level, why a particular design requires a longer time to execute than another; this situation is analogous to someone trying to determine why the assembly language translation of one program is different from another.

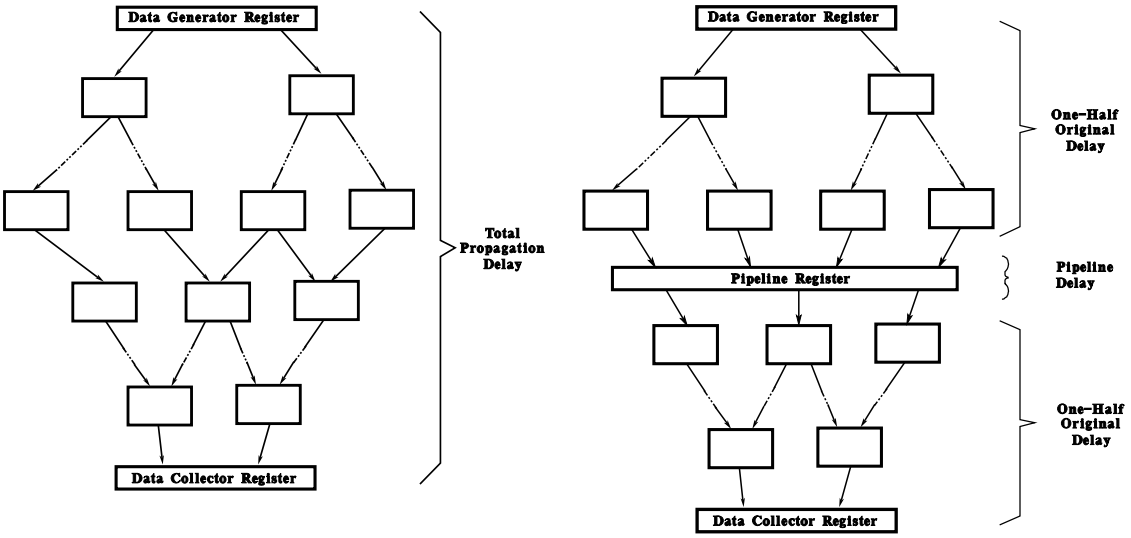


Figure 5.1: A non-pipelined vs a pipelined DFG

A more general solution is to pipeline the ILB. An illustration of a non-pipelined vs pipelined ILB is shown in Figure 5.1. Without pipelining, the ILB propagation delay includes that of all of the nodes between the data buffering registers within the data generator and collector nodes. The pipelining optimization adds registers to strategic places within the ILB, so that intermediate results can be computed and stored temporarily. These intermediate results require a shorter sequence of circuitry to compute when compared to the original computation, and hence will have a shorter propagation delay. The overall speed of the circuit is determined by the maximum delay between registered values, so if the pipeline registers are placed so that the propagation delay of each shorter segment is one-half of the original, a circuit that is pipelined can run up to twice as fast as one that is not. The resulting ILB will now take two clock cycles to produce a final result, but the clock period need only be about half as long (the registers add some propagation delay themselves, so the final clock period will never quite reach half). The additional cycles required to perform the full computation only affect startup delays, not steady-state throughput, since several computations can be in different sections of the pipeline at once.

In most cases, the clock speed of the entire circuit is determined by the propagation delay of the ILB, since it usually contains the largest sequence of computations within the final design. However, for very short ILBs, this may not be the case. The top and

bottom registers shown in Figure 5.1 are part of the data generator and collector modules, respectively. These modules also have a propagation delay - this delay depends on the number of buffer registers that must be instantiated for a given design and the complexity of the interconnection between these buffers. For typical problems implemented on the Wildforce board this delay usually falls in the range of 30 to 37 nanoseconds, corresponding to a maximum frequency of 27 to 33 MHz. These values represent the minimum delay (and thus maximum operating frequency) of any design.

The pipeline registers must be placed at the proper places in the ILB to be effective. One method for implementing pipelining involves critical path analysis. Within a combinational circuit, one or more signal paths from top to bottom requires the longest propagation delay – this is the *critical path*. If registers are placed in the "middle" (in terms of propagation delay) of the critical path, the delay will be cut approximately in half. If the circuit is now re-analyzed, a new critical path can be found. If the propagation delay of this path is still too long, the register placement process can be repeated. This iterative process is repeated until a propagation delay criteria is met, or until a maximum number of registers have been reached, or until the circuit runs out of space due to register placement.

A modification of the above approach places a register in the critical path, and registers in each of the other (shorter) paths at a position where the delay of each half of the resulting split path is less than the delay produced by splitting the critical path. This method reduces the iterations required to finish. Multiple pipeline stages can be produced with this method by placing the first set of registers in the critical path at intervals of one-third, or one-fourth, etc. of the total propagation delay, and then placing registers in the other paths at points such that the delay of each piece does not exceed the delay through the corresponding piece along the critical path. In order to synchronize the values that are being computing, it is important to place pipeline registers in *every* path of the ILB. Stated another way, a value that propagates through the ILB must "touch" the same number of pipeline registers, regardless of the path taken through the ILB.

These methods assume that information concerning propagation delays is available.

Unfortunately, the exact propagation delays through the circuit are not known until after synthesis and place-and-route. Since these steps take minutes or even hours, the amount of time required to determine where pipeline registers should be placed can become prohibitive.

A more convenient place to perform the critical path analysis is at the dataflow graph level, before time is spent on synthesis and place-and-route. The compiler typically produces a DFG within a matter of seconds, and the critical path analysis could be performed quickly on the DFG structure. This would add only a small amount of time to the entire process. However, no propagation delay information, relative to the DFG, is currently available. On the other hand, since each DFG node is translated into an equivalent piece of VHDL, it may be possible to come up with an estimate of the propagation delay for each DFG node, which is accurate enough to estimate proper pipeline register placement.

The next section describes the process used to develop a propagation delay estimation model for each DFG node. Section 5.3 describes how this model is used to estimate the placement of pipeline registers within the ILB. Section 5.4 presents the results of the pipeline register placement, showing both the accuracy of the estimation model and the effectiveness of the pipeline registers in reducing the propagation delays on several SA-C codes. Finally, Section 5.5 discusses improvements that can be made to the estimation model; some of these improvements are motivated by insights gained from research that was done to improve the accuracy of a similar model that estimated circuit space.

5.2 Estimating the Propagation Delays of the Dataflow Graph Nodes

This section describes the process used to determine estimates of the propagation delays for each of the nodes that can appear in a DFG.

As described earlier, a DFG is translated to hardware in several steps: first, the DFG is translated into VHDL by a tool developed as part of the Cameron project. Secondly, the VHDL is synthesized into a device-dependent netlist, which for our hardware consists of an interconnection of FPGA CLBs and I/O blocks. Finally, this netlist is place-and-

routed, and converted into a final binary configuration. Both the synthesis tool and the place-and-route tools can produce reports that specify the propagation delays through the circuit.

Our approach toward developing an estimation model that operates at the DFG level is rather straight-forward - we take a "black box" approach, where DFG nodes are translated into VHDL, synthesized and place-and-routed. Then the timing information produced by these tools is related back to the DFG nodes.

This approach presents some significant challenges. Most of the original form of the program has been abstracted away at this level - the design has been flattened, and almost none of the original interfaces between the DFG nodes still exists. Both the synthesis and the place-and-route processes perform optimizations, causing some of the logic specified by the VHDL to be optimized into a considerably different form, possibly even dissolving it away completely. This makes it difficult to recognize the correspondence between the original VHDL and the resulting circuit. The synthesis and place-and-route tools are proprietary - the vendors of these tools do not wish to divulge the exact optimizations and algorithms used to perform the optimizations, except in general terms. Finally, the place-and-route process is at least partially non-deterministic - the "answer" may be different for identical executions of the same problem!

Fortunately, the design of the DFG-to-VHDL translator provides a means of tracing each signal back to the DFG. Each intermediate VHDL signal created during translation is given a name that is derived from the number of the DFG node that produced the signal. The output signals from each node are given a name derived from the node type, the node number, and output number of the node. So, for example, the first output (output zero) of an unsigned add (`UADD`) node that happens to be node number 18 in a particular graph would have the name `UADD18OUT0`. A more complete example of the use of these derived signal names was shown in Figure 4.5. For those signals that have not been dissolved away by optimization, the signal name appears in the netlist produced during synthesis, and in the final place-and-route report. The cumulative propagation delay of the circuitry leading up to the signal can be found in the accompanying timing reports.

For the propagation delay model, the results reported by the synthesis tool (Synplify) were used, rather than the (admittedly) more accurate place-and-route results. There are several reasons for this decision. First, the place-and-route process is in part nondeterministic, making it possible to obtain different delay values for different runs of the same problem. Thus, in order to get reasonably accurate delay values, several runs be would need to be made, and the results averaged to obtain final numbers. Since each place-and-route run can take an hour or more, a considerable amount of time would be required to use place-and-route estimates. By contrast, Synplify only takes a few minutes for each run, and produces the same results each time.

To further justify this decision, it is useful to analyze the difference in accuracies produced during synthesis vs. place-and-route. Synthesis produces estimates of propagation delay which account for delay through the CLBs, but not delays caused by routing. The estimates are therefore smaller than those produced by the full place-and-route process, but do correspond in general with the complexity of the resulting design. Experience has shown that the propagation delays estimated by Synplify are 20 - 30% lower than the actual delays produced after place-and-route. This seems like a rather large discrepancy; however, the ratio is relatively consistent for all designs (that is, one can reasonably predict the final propagation delay of the design by adding 30% to the numbers reported by Synplify, and typically be within a small few percent of the actual design speed).

For the purpose of pipelining, the *ratio* of the propagation delay at a particular point in the circuit to the total propagation delay of the circuit is required. Thus relative, rather than absolute, numbers are more important. Since the ratios between synthesis results vs. place-and-route results are consistent, we assume the numbers derived from Synplify will produce a model that is "close enough."

The process for deriving propagation delay values for the model proceeds as follows. First some very simple SA-C programs which produce DFGs that have a nearly one-to-one correspondence between operation and DFG node, such as `ADD`, were constructed and synthesized. Next, other slightly more complicated programs were created. In these examples, the DFG still contains most of the original nodes, although not necessarily in a

one-to-one correspondence. For example, programs containing an `if` statement produce two separate DFG nodes – a compare node such as `UGT` (unsigned greater than) and a `SELECT` node, to choose the correct value determined by the compare. Finally, programs that generate more complicated node sequences were also studied, to assess the situation where the synthesis process optimizes the design by combining some operations into an equivalent boolean function, and dissolves away many of the signals that appear in the original DFG.

Using the "technology view" function of Synplify, which provides a schematic view of the synthesized netlist, signal names were traced by hand throughout the circuit. Propagation delay estimates appear next to each signal, making it possible to determine the delay from input signals to output signals of those nodes which haven't been dissolved away or combined with other nodes. In those cases where a node or signal has been dissolved, we try to find another example to analyze where a dissolve does not occur. In the cases where nodes have been combined, it is possible to compute the delay of a larger sequence of nodes, and then distribute the cumulative delay between the individual nodes in the sequence. The final value determined for a given node involves a "best guess" derived from inspection of delay values obtained from the methods described above.

Since the manual process is prone to error, the delay values derived from the analysis were recorded in a data file. This makes it easy to update the numbers when inaccuracies are discovered, without having to modify and recompile the actual model code.

The technology view creates mechanically-generated schematics that are not particularly straight-forward to analyze. The circuitry comprising a single node may span several pages of schematic. As mentioned before, the synthesis process destroys most of the original structure of the program. The complete schematic often occupies tens or even hundreds of pages, so the sheer mass of information makes the analysis difficult. However, after studying a large number of programs, some patterns begin to emerge.

Most nodes can be modeled with a linear graph as shown in Figure 5.2. The line consists of a fixed delay portion plus a variable portion that is a function of the bit width of the input value(s) of the node. Increasing bit width generally has a greater effect on

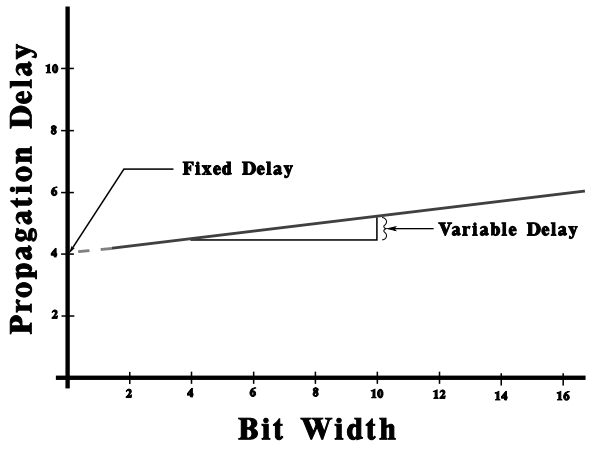


Figure 5.2: Linear graph of bit-width vs propagation delay

the size (i.e., the number of configurable logic blocks, or CLBs) of the design, and a lesser effect on the propagation delay, since the CLBs are connected in parallel. Nonetheless, more bits contribute to a more complicated "sideways" interconnect between the otherwise parallel-connected CLBs. For example, since a CLB consists of two four-bit inputs, an 8-bit adder will require 2 CLBs, with signals connected between them to implement the carry logic. Thus, the slight upward slope of the line is caused by the added propagation delay introduced by this extra wiring.

The DFG nodes that implement the "reduction" operators of SA-C, such as the `USUMMANY` (unsigned sum–many), provide an additional parameter - the number of inputs. The first implementation of the estimate model used the linear model, where the slope was determined by the product of the number of inputs and their bit width. This proved to be too simplistic for these reduction cases.

The reduction nodes have been implemented as a binary tree, rather than the simpler implementation (Patel 2000). The difference is illustrated in Figure 5.3 - the left side shows a linear implementation, while the right side illustrates a binary tree implementation. This implementation generally causes the propagation delay to increase as the $\log_2$ of the number of inputs; however, it actually increases in steps, where each step occurs at each integral value of $\log_2$ (i.e., 2, 4, 8, etc.), since another "layer" of logic is added at these points. Once a new level is introduced, the propagation delay behaves much like the

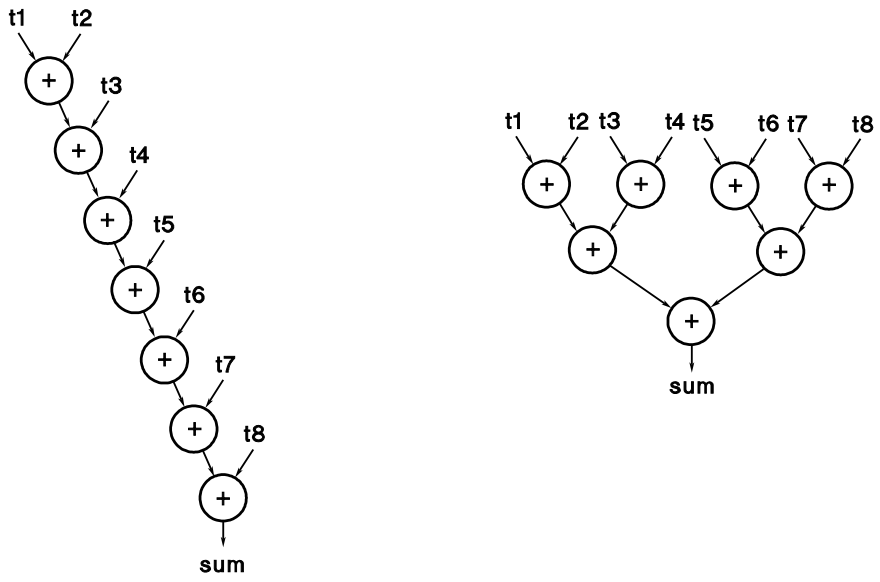


Figure 5.3: Linear vs binary tree implementation of the `USUM-MANY` node

simpler linear model for that level. Therefore, the propagation delay for these nodes is implemented as a "family" of curves, one for each level. The particular curve within the family is selected by the function $int[\log_2 n]$, where n is the number of inputs. A graph of a typical family of curves, with the actual function used to predict the propagation delay shown in bold, is shown in Figure 5.4.

Finally, while most nodes are combinational, and therefore contribute propagation delay to a single segment of circuitry, some nodes contain registers. These nodes serve as the termination of one propagation delay segment, and the initiation of another segment; they contribute delay to both segments. Examples of these nodes include the generator nodes, the collector nodes, and the register nodes that will be added by the pipelining algorithm described later. Such nodes partition the DFG into several separate timing segments - the overall circuit speed is determined by the largest propagation delay of any of the segments.

In the model, these nodes contain two sets of propagation delay numbers - one for the clock cycle controlling the inputs, and one for the outputs. In the current system, no hardware-generating nodes appear before the generator nodes or after the collector nodes - by definition, these two types of nodes form the beginning and end of the ILB.

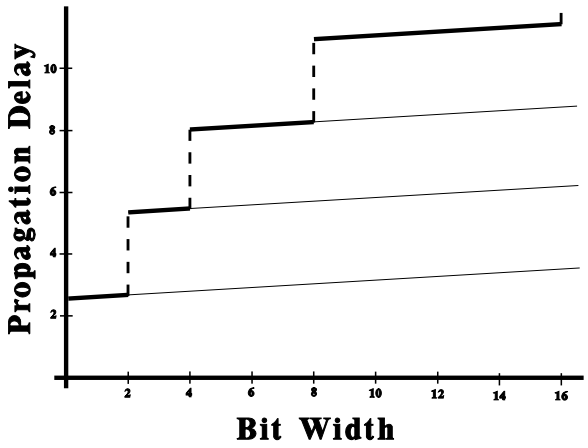


Figure 5.4: A "family of curves" graph, characteristic of a binary tree implementation of the `MANY` nodes.

For the generator nodes, the propagation delay recorded for the "input" values actually represents the entire delay through the VHDL components that form the generator. For the collector, the delay recorded for the "output" values represents the delay through the VHDL that form the collector.

Presently, the delays for the generator inputs are longer than the collector outputs, and therefore determine the maximum operating frequency of the circuit, about 30 MHz. In the estimation model, if the total propagation delay of the ILB (which includes the output delay of the generators and the input delay of the collectors) is less then the delay of the generator inputs, then the generator input delay will determine the circuit operating frequency, and any pipelining of the ILB will not improve the performance. Therefore, this value is used to establish the minimum point at which pipelining can be effective.

Table 5.1 shows the numbers used for the simple, strictly combinational nodes. For those nodes that represent multiple cycles, table 5.2 lists the values for the propagation delay contributed by the input signals and output signals. Finally, table 5.3 shows the numbers for those nodes that are modeled with a family of curves.

5.3 Pipelining the Inner Loop Body

Given a model to estimate the propagation delay of each DFG node, the pipelining process can proceed as follows: after the DFG is created by the SA-C compiler, the DFG file is

Node Name	Description	Model Parameters	
		Fixed	Variable
Unsigned Arithmetic Nodes			
UADD, USUB	Unsigned Arithmetic	9.1	0.0
ULT, ULE, UEQ, UNEQ, UGT, UGE	Unsigned Compares	9.1	0.0
UMUL	Unsigned Multiply	36.8	1.9
UMEDIAN-MANY, VAL-AT-FIRST-UMAX-MANY, VAL-AT-FIRST-UMIN-MANY, VAL-AT-LAST-UMAX-MANY, VAL-AT-LAST-UMIN-MANY	Single-Value Reductions	5.0	0.0
ACCUM-USUM-VALUES, ACCUM-UMAX-VALUES, ACCUM-UMIN-VALUES	Multi-Value Reductions	5.0	0.0
Signed Arithmetic Nodes			
IADD, ISUB	Signed Arithmetic	9.1	0.0
ILT, ILE, IEQ, INE, IGT, IGE	Signed Compares	9.1	0.0
IMUL	Signed Multiply	38.8	1.9
NEGATE	Negate	9.1	0.0
SQRT	Square Root	10.0	18.6
IMEDIAN-MANY, VAL-AT-FIRST-IMIN-MANY, VAL-AT-LAST-IMIN-MANY, VAL-AT-FIRST-IMAX-MANY, VAL-AT-LAST-IMAX-MANY	Single-Value Reductions	5.0	0.0
ACCUM-ISUM-VALUES, ACCUM-IMAX-VALUES, ACCUM-IMIN-VALUES	Multi-Value Reductions	5.0	0.0
Bit/Boolean Operations			
BIT-AND, BIT-OR, BIT-EOR, BIT-COMPL	Bitwise Operators	9.1	0.0
L-SHIFT, R-SHIFT	Shift (Binary Multiply)	0.0	0.0
BIT-SELECT, BIT-CONCAT	Simple Bit Operators	5.0	0.0
ACCUM-AND-VALUES, ACCUM-OR-VALUES	Bitwise Reductions	5.0	0.0
CHANGE-WIDTH, CHANGE-WIDTH-SE	Bitwidth Change	0.0	0.0
Miscellaneous Operations			
INPUT	Input Parameter	4.7	0.0
OUTPUT	Output (Finish)	4.1	0.0
CHOOSE	Conditional value selest	5.0	0.0
SELECTOR	Multiple Choose	5.0	0.0
CIRCULATE	Loop carried value	5.0	0.0

Table 5.1: Nodes that have simple, single-valued model parameters

Node Name	Description	Model Parameters			
		In Cycle		Out Cycle	
		Fix	Var	Fix	Var
SCALAR-1D-GENERATOR, SCALAR-2D-GENERATOR	Single Value Generators	54.1	0.0	12.0	0.0
ELEMENT-1D-GENERATOR-*n*-BIT, ELEMENT-2D-GENERATOR-*n*-BIT	Single Array Element Generators	54.1	0.0	12.0	0.0
WINDOW-1D-GENERATOR-*n*-BIT, WINDOW-1D-GENERATOR-*n*-BIT	Window (Sub-array) Generators	54.1	0.0	12.0	0.0
WRITE-TILE-1D-1D-*n*-BIT, WRITE-TILE-1D-2D-*n*-BIT, WRITE-TILE-2D-1D-*n*-BIT, WRITE-TILE-2D-2D-*n*-BIT	Collector (write) Nodes	9.1	0.0	29.1	0.0
REGISTER, REGISTER-MANY	Pipeline Registers	4.1	0.0	4.5	0.0

Table 5.2: Nodes that represent multiple-cycle operations

Node Name	Model Parameters									
	Family 1 1-2 inputs		Family 2 (3-4 ins)		Family 3 (5-8 ins)		Family 4 (9-16 ins)		Family 5 (17-32 ins)	
	Fix	Var	Fix	Var	Fix	Var	Fix	Var	Fix	Var
USUM-MANY	20.40	0.15	26.50	0.15	32.60	0.15	38.70	0.15	44.80	0.15
ISUM-MANY	20.40	0.15	26.50	0.15	32.60	0.15	38.70	0.15	44.80	0.15
UMIN-MANY	24.56	0.25	34.53	0.50	44.49	8.75	54.46	1.01	64.42	1.26
IMIN-MANY	11.46	0.25	21.43	0.50	31.39	8.75	41.36	1.01	51.32	1.26
UMAX-MANY	24.56	0.25	34.53	0.50	44.49	8.75	54.46	1.01	64.42	1.26
IMAX-MANY	11.46	0.25	21.43	0.50	31.39	8.75	41.36	1.01	51.32	1.26
AND-MANY	5.67	0.10	6.70	0.01	9.56	0.10	10.66	0.10	13.46	0.10
OR-MANY	18.76	0.10	19.86	0.10	22.66	0.10	23.76	0.10	26.56	0.10

Table 5.3: Nodes that are modeled as a family of curves

input to the pipeline program. This graph is annotated with the propagation delay of each node. The cumulative propagation delay at each node input and output is computed, and the critical (longest) path is determined. Registers are first placed in the critical path, then in all other paths through the DFG. A sanity check is made to insure that all paths from top to bottom pass through the same number of pipeline registers, since the timing of all signals must be consistent. Finally, the new DFG, now containing registers, is written out to a file.

To add multiple stages of pipelining, the pipeline program can be used iteratively. During the first iteration, the registers are placed more than halfway down the critical path; for example, to add three pipeline stages, the registers would be placed two-thirds of the way down the critical path. Subsequent iterations add additional registers at points further up the graph; in the case of the two stage problem above, one additional iteration would be used to place registers at one-third of the propagation delay along the critical path.

The newly created DFG can then be processed in the normal way by the DFG-to-VHDL translator. The translator has been modified to handle the new `REGISTER` (and `REGISTER-MANY`) nodes. Additionally, a delay circuit must be added to the `StoreData` signal, which controls the operation of the `Collector` circuits and the bottom of the DFG. For each pipeline stage added, this signal must be delayed by one cycle.

Figure 5.5 shows compilation process, modified to accommodate pipelining. Because of the modularity of the compilation system, relatively minor modifications are necessary to add pipelining to the compilation and translation process.

5.4 Pipelining Performance

To evaluate performance, the pipelining algorithm was run on a representative suite of SA-C programs. Test cases were run to verify both the values in the propagation delay model and the overall effectiveness of pipelining.

Table 5.4 compares the results of the estimated propagation delay with the actual values as reported by the Synplify VHDL compiler. Also shown are the number of DFG

Benchmark Name	No of Nodes	ILB Propagation Delay		
		Actual	Estimate	% Error
MajThresh	45	41.8	51.1	22.27%
Add8	19	32.2	35.2	9.2%
AddS_opt	32	35.6	40.2	12.9%
Convolve2D8	92	87.3	76.1	-12.7%
Conv2D8-strip_4_3	159	87.9	76.1	-13.4%
Conv2D8-strip_5_3	226	87.9	76.1	-13.4%
ConvSep2D8	236	150.3	137.8	-8.3%
CrossCorr	32	35.9	47.2	31.5%
Gaussian3_8	45	44.1	43.1	-2.4%
Gauss3_8-strip_4_3	74	44.9	43.1	-4.0%
Gauss3_8-strip_5_3	99	43.4	43.1	-0.9%
Gauss3_8-strip_6_3	124	50.3	43.1	-14.4%
Multiply8	18	45.4	46.9	3.2%
MultScale8	68	122.7	106.1	-13.5%
NormCrossCorr	248	576.8	605.5	5.0%
dilation	74	64.5	58.6	-9.2%
erosion	83	64.7	58.6	-9.4%
waveletl1	175	81.1	101.5	25.1%
RGBtoXYZ	44	47.7	66.9	40.2%
XYZtoRGB	59	48.3	59.6	23.4%
gaussianfilter	20	34.5	62.7	81.8%
laplacefilter3x3	29	44.9	37.0	-17.7%
laplacefilter5x5	58	49.3	69.0	39.8%
robertsmag	29	214.8	309.0	43.9%
sobelmag	40	305.3	383.3	25.6%
mp4	20	27.8	18.7	-33.0%
prewittmag	36	279.4	359.8	28.8%
prewittmagsqrt	188	275.6	385.1	39.7%
			Average Error	**10.97%**
			Avg Absolute Error	**20.27%**

Table 5.4: Comparison of actual vs. estimated propagation delays for a representative set of SA-C benchmark programs

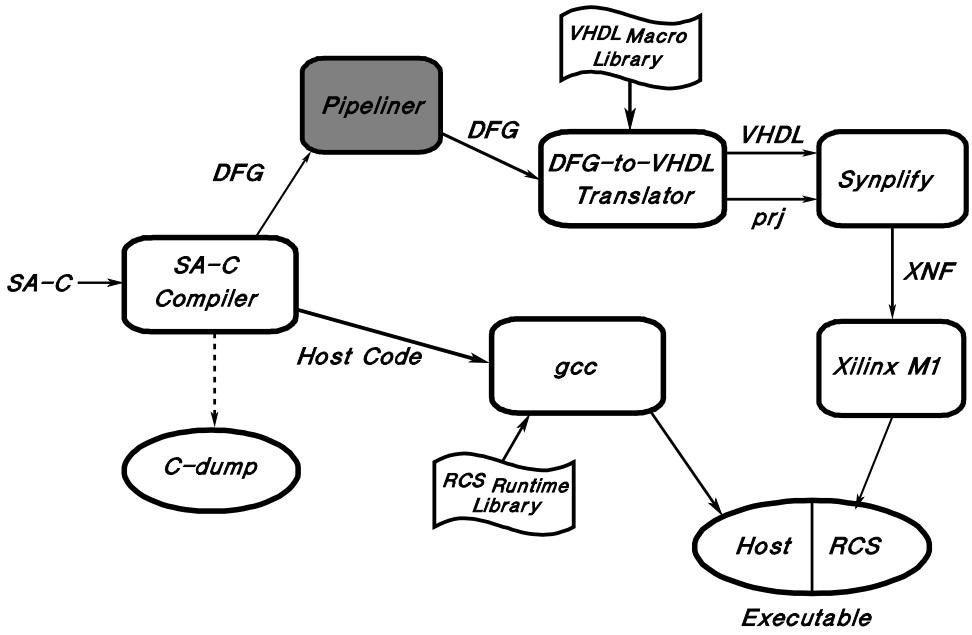


Figure 5.5: SA-C compilation process, modified to include pipelining of the DFG

nodes included with each program, which is a rough measure of the size of the program. The estimated results for some programs are less than actual, while others are more than actual; this is due to the use of a large number of test cases to develop the delay values for each node, where sometimes the node is completely dissolved away, and sometimes it is incorporated into a larger sequence of logic where the original node is no longer visible. The averaging of the cases attempts to capture the effects of the synthesis optimizations, and causes the estimates to be either larger or smaller than actual.

The average error for the programs shown in Table 5.4 is 10.97%, and the average of the absolute values of error slightly exceeds 20%. It is clear that the aggressive optimizations performed by Synplify make it difficult to arrive at estimates that are extremely accurate in absolute terms. Similar results are reported when attempting to estimate the space requirements of DFG nodes in (Kulkarni, Najjar, Rinker, and Kurhadi). Section 5.5 below discusses some improvements that were made to the space estimation model to improve accuracy.

While the estimates seem to be rather inaccurate in *absolute* terms, what is most important for pipelining is that the *relative* estimate be close enough to specify proper

	Benchmark Name	Original Design Size FMAPs	Original Design Size Regs	Original Design Clk (MHz)	Pipelined Design Size FMAPs	Pipelined Design Size Regs	Pipelined Design Clk (MHz)	Percent Incr Reg	Percent Incr Clock
1	Conv2Dstrip5_3	4871	627	11.4	4989	984	22.3	56.94%	95.61%
2	Conv2Dstrip4_3	3255	466	11.4	3382	713	22.3	53.00%	95.61%
3	Convolve2D8	1656	325	11.5	1721	439	22.4	35.08%	94.78%
4	NormCrossCorr	7685	1031	1.7	7963	1061	3.3	2.91%	94.12%
5	ConvSep2D8	4560	467	6.7	4565	505	12.9	8.14%	92.54%
6	waveletl1	1370	656	12.3	1357	759	21.0	15.70%	70.73%
7	XYZtoRGB_1	1080	393	20.7	1030	562	31.6	43.00%	52.66%
8	RGBtoXYZ_1	1118	412	21.0	1092	580	31.6	40.78%	50.48%
9	Multiply8	368	170	22.0	367	174	31.0	2.35%	40.91%
10	Gaussian3_8	334	204	22.7	331	217	31.0	6.37%	36.56%
11	sobelmag	1155	219	3.3	1146	233	4.5	6.39%	36.36%
12	laplacefilter3x3	405	197	22.3	372	273	29.6	38.58%	32.74%
13	robertsmag	716	169	4.7	714	187	6.2	10.65%	31.91%
14	prewittmag	1035	227	3.6	1022	231	4.7	1.76%	30.56%
15	dilation	508	285	15.5	507	360	19.7	26.32%	27.10%
16	laplacefilter5x5	728	360	20.3	700	590	25.7	63.89%	26.60%
17	erosion	523	285	15.5	519	361	19.4	26.67%	25.16%
18	prewittmagsqrt	1017	220	3.6	1310	237	4.5	7.73%	25.00%
19	MajThresh	299	228	23.9	294	238	28.1	4.39%	17.57%
20	Gauss-strip_4_3	624	327	22.3	615	350	25.4	7.03%	13.90%
21	MultScale8	440	169	8.2	499	189	9.3	11.83%	13.41%
22	gaussianfilter	308	192	29.0	308	265	30.3	38.02%	4.48%
23	Gauss-strip_6_3	1059	523	19.9	1056	572	20.7	9.37%	4.02%
24	Gauss-strip_5_3	869	445	23.0	844	480	23.4	7.87%	1.74%
25	Add8	261	157	31.0	277	167	31.0	6.37%	0.00%
26	AddS_opt	438	278	28.1	439	299	28.1	7.55%	0.00%
27	CrossCorr	335	216	27.9	343	240	27.9	11.11%	0.00%
28	mp4	165	100	35.9	165	116	35.9	16.00%	0.00%

Table 5.5: Original size (in terms of CLB FMAPs and Register bits) and clock speed vs. pipelined size and speed for the SA-C benchmark programs

placement of the pipeline registers. Even if the placement is not optimal, a design should benefit from pipelining.

Table 5.5 shows a comparison of performance between the original non-pipelined version and the pipelined one for each of the benchmarks. Of main concern in this table is the change in clock speed between the non-pipelined and pipelined results - the last column in the table shows the percentage increase in clock speed for each benchmark. The entries in the table are ordered from best to worst.

Table 5.5 also shows the amount of FPGA resources required by each benchmark, in terms of FMAPs and CLB registers (HMAPs are deemed not a factor in any of the size

considerations, so are not reported here). Since the pipelining process adds registers to the design, there was some concern about the amount of additional FPGA space required to do the pipelining. As Table 5.5 shows, this appears to not be a significant concern in most cases. In terms of FMAPs, only small increases occur, and in many instances, the number actually goes down. Similarly, the number of registers required to implement pipelining is a fairly small percentage, although some of the benchmark do require a 50 - 60% increase.

The first several benchmarks listed in the table achieve a clock increase of better than 90% – this indicates a near optimal placement of the pipeline registers in the DFG, also suggesting that the propagation delay estimates are accurate enough in a relative sense to produce good results. Note that since the pipeline registers themselves introduce some propagation delay overhead, a "perfect" increase of 100% will never occur. Also, the true midpoint of propagation delay might occur in the middle of a DFG node, requiring that the pipeline register be placed on one side or the other of the middle node.

It is also worth noting that many of the first benchmarks in the table are also the largest, both in terms of number of DFG nodes and in terms of FPGA resources required. This would suggest that the "averaged delay" methodology used in the estimation model might work reasonably well when there are a large enough number of nodes that the inaccuracies inherent in the averaging technique might tend to cancel each other out. In general, these largest benchmarks can benefit the most from pipelining.

Several of the benchmarks achieve only a slight increase (or no increase) in overall clock speed, because the ILB is already so fast that the clock speed is determined by the data generator node. In these cases, pipelining causes the benchmark to increase in speed up to the maximum clock rate allowed by the particular data generator - their pipelined clock rates are shaded in Table 5.5.

The remaining benchmarks do not achieve the increase in performance that is expected, although they all exhibit some increase. It is likely that this is due to inaccuracies in the propagation delay estimations for these benchmarks - several of these benchmark display considerable estimate inaccuracies in Table 5.4. It is expected that they would benefit

from improvements to the estimation model, which is discussed in the next section.

5.5 Improvements to the Estimation Model

The estimation model described in the previous sections provide propagation delay estimates that are usually within 20% of actual when using the benchmark programs; however, several benchmark estimates contained errors of up to 40% as compared with the actual. This section discusses the possible causes of the model errors, along with some suggestions for changes to the model to mitigate these errors.

As an offshoot of the development of the DFG *timing* estimation model described in this dissertation, a *space* estimation model for DFGs implemented on FPGAs was developed as a separate project. This effort is described fully in (Kulkarni 2001) and (Kulkarni, Najjar, Rinker, and Kurhadi). This estimation model includes a detailed analysis of sources of errors in the model. The goal of the space estimation project is somewhat different than the time estimation model described here; nonetheless, many of the techniques used in the space estimation model could be applied as improvements to the time estimation model, and many of the findings provide insights to the estimation model.

The space estimation model began by using the same basic framework as in the estimation of the time model. The following sources of errors were identified while analyzing the model results:

- The actual recording of data is done using small graphs that were specifically designed to isolate individual node types. This simplifies the collection of data on individual nodes, but it doesn't take into account the interactions that occur among nodes. These interactions are caused by the aggressive optimizations that Synplify performs while compiling the VHDL code that results from the DFG-to-VHDL translator. Many combinations of CLBs are replaced with smaller, more efficient sequences. Since Synplify is not constrained in its optimizations by DFG node boundaries (in fact, it doesn't know anything about DFG nodes), some of the combinations to be replaced occur between CLBs produced by different DFG nodes. The resulting timing estimates will not account for these inter-node optimizations.

- Some nodes, when scaled, don't behave like the simple linear model (or even the linear model with steps) described earlier. For example, the size of a multiply node increases as the square of the inputs. For the most part, the timing of such nodes is not as dramatically affected as the space (for example, parallel circuits have a much greater effect on size than on speed), but there is an effect, nonetheless.

- Differences occur in the synthesis of some node structures caused by the level of optimization *effort* performed by Synplify. That is, Synplify often trades time for space (or vice versa) during synthesis. Some of the choices can be specified by compiler options to Synplify, but others are selected automatically as it attempts to meet design requirements. The simple graphs designed to facilitate gathering the space/time parameters may not require the most aggressive optimizations, and therefore the collected data does not accurately predict actual behavior on more complex designs. Worse yet, one design that includes a certain node or node sequence may not require the higher degree of optimization to meet clock speed targets as does a more complicated design containing the same node or node sequence. This results in the real propagation delays being different for the same nodes in two different designs.

- Different versions of Synplify appear to produce slightly different designs. Thus, the data collected by analyzing the circuit produced by a previous version of Synplify may not be accurate for the latest version. New updates are issued regularly (every few months), so data collected a few months ago may not reflect results from the most recent version of Synplify.

- Since the extraction of the space and time values is done manually, by inspecting reports generated by Synplify, there is the possibility of errors in recording the data.

The improved space estimation model tries to mitigate some of these problems using some modifications from the original model. First, several different approximation formulae are used instead of just one. These include:

- $Y = constant$ – The node contributes a constant amount of space to the circuit.

- $Y = p_0 + p_1 \times x$ – The node contributes space that is a linear function of the bit width x.
- $Y = p_0 + p_1 \times (x - p_2)^2$ – The node contributes space that is a quadratic function of the bit width x.
- $Y = p_0 + (x - p_1) \times (z - p_2)$ – The node contributes space that is a function of bit width x and number of input values z.
- $Y = p_0 + (p_1 * \frac{x}{2})(\frac{z}{2} - 1)$ – The node contributes space that is a multi linear function of the bit width x and the number of inputs z.

At first glance, this model seems considerably more complicated than the simpler time estimation model described previously. At least some of this is due to the nature of space estimation versus time estimation. For example, the first two equations can be implemented directly with the simpler time model. Estimates for the `MULT` nodes use the quadratic equation to estimate space. In terms of propagation delay, these nodes are mostly a linear function of the width of inputs, so they can be reasonably estimated with the simpler model. As the size of inputs increase, the propagation delay appears to go up slightly faster than linear. This is due to the additional CLBs required to perform intermediate combinations as the bit width increases. Thus, the simple model tends to underestimate the actual delay. Further study is necessary to determine if this additional delay follows a regular pattern that could be added to the simple model.

Finally, the last two equations cover nodes that are modeled with the "stair step" function in the time model. Such nodes increase in space as a linear or multi linear function of the number of inputs, but mostly as $\log_2$ in terms of time, as described previously. The parallel cross-wiring required by these nodes does contribute some additional amount to the propagation delay; this is not accounted for in the present model, except in a rather coarse fashion. On the other hand, this additional delay is somewhat difficult to characterize, since the relationship between the propagation delay and number of inputs (that is, parallelism) is dependent at least partially upon the success that the place-and-route software has in synthesizing the final circuit configuration. The algorithm for doing

this is at least partially heuristic.

In summary, the effort involved in more accurately characterizing space requirements was valuable in better understanding the time model. The use of several different approximation equations helped to improve its accuracy. While not all of the new equations are appropriate, in general the techniques developed for space estimation should also improve time estimates.

A second technique that the space model uses to improve its estimate is to identify some of the combination sequences that Synplify optimizes, and then make similar substitutions for the space estimates in the model. Several of these sequences were discovered while developing the model – a couple of examples include:

- Multiplication by a power of two - the expensive multiply operation is replaced with a simpler shift operation.
- Simplification of the calculations required to perform comparisons. Often, the full value involved in a comparison is not required to determine the outcome of the comparison, so it can be simplified.

Since this technique involves substituting a shorter *sequence* for a longer one, this will also reduce the propagation delay. As a consequence, it is expected that adding similar substitutions to the simpler time model will also improve its accuracy.

Chapter 6

Using Lookup Tables to Implement Dataflow Graph Segments

This chapter explores another method for reducing the propagation delay of complex inner loop bodies. The basic idea is to replace a portion of the inner loop body with a faster lookup table implementation. In many cases, this technique will improve the speed, but at the expense of requiring more space on the reconfigurable hardware. The technique has the potential of significant speedups in some cases, although it appears that not all DFGs exhibit characteristics that will benefit from the technique. Thus, its application will not be as general as the pipelining technique described in the previous chapter.

This chapter first describes the general characteristics required in order for the lookup table replacement to be effective. Next, the implementation of lookup tables using FPGAs is discussed so that their performance can be better understood. Then two different techniques for discovering opportunities for applying lookup tables are discussed in this chapter; the first algorithm uses a variation of the Ford Fulkerson maximal flow algorithm, providing a "global mincut" approach. The second algorithm uses a "flood" algorithm to discover more localized subgraphs within the DFG that can be replaced with lookup tables.

6.1 Introduction

The DFGs generated for some applications exhibit similar characteristics, as pictured in Figure 6.1. At the top of the DFG the connections fan out into several nodes; this is

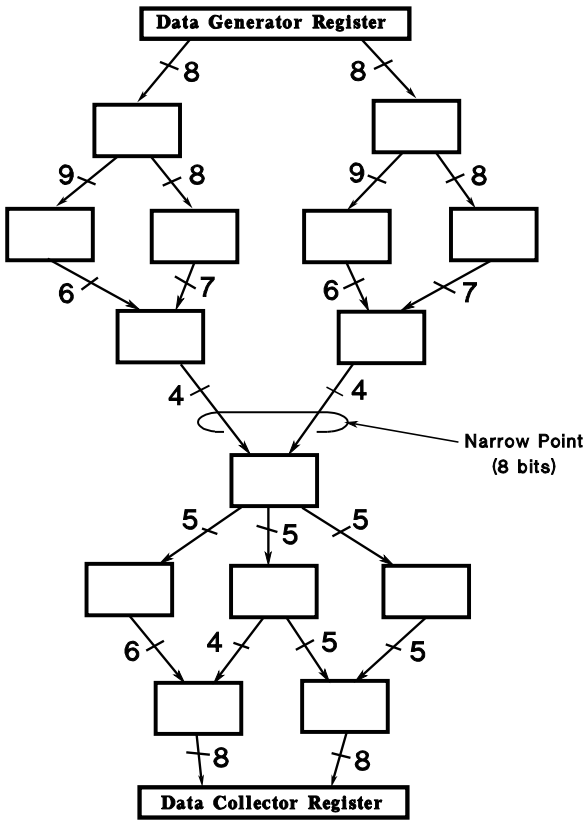


Figure 6.1: Dataflow graph showing a narrow point in the graph

indicative of parallelism in the application. However, the graph then narrows down to a small few number of nodes, with inputs consisting of a small number of bits. The graph then fans out again before it terminates in a small number of values that are to be written as results.

One cause for such a graph shape is an application that invokes a complex subroutine in the middle of the inner loop. If the arguments to the subroutine are few in number, a narrow point occurs. By the time a program is converted to a DFG, however, the subroutine is in-lined, and therefore loses its identity within the graph - only the artifact caused by the argument interface remains.

For example, consider the Prewitt algorithm described in several places earlier. A manually-generated DFG was pictured in Figure 3.2. In this picture, the DFG narrows just before the `sqrt` function. The automatically-generated DFG exhibits the same overall shape, although the start of the `sqrt` routine is not easily distinguishable from the other

calculations in the DFG.

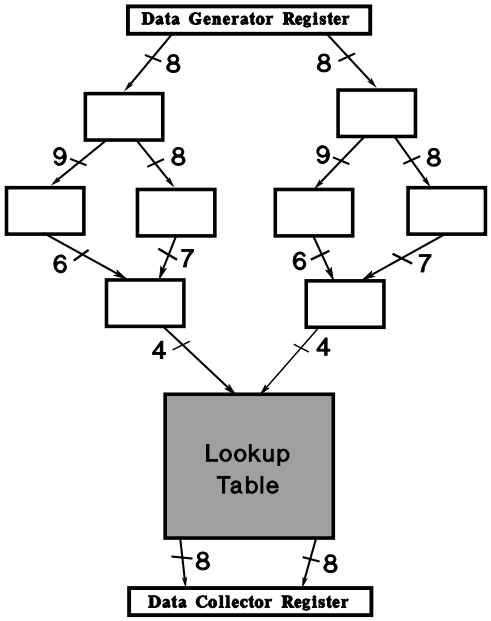


Figure 6.2: Dataflow graph with the lower portion replaced with a lookup table

These narrow points provide potential opportunities for the use of lookup tables. The process proceeds as follows: the narrow point is identified in the DFG, and that portion of the DFG below the narrow point is replaced with a lookup table. The lookup table, whose values are generated from that part of the DFG that was "cut out," provides the output values that originally were computed by the cut out portion of the DFG. The result of replacing part of the DFG with a lookup table is shown in Figure 6.2. If the propagation delay of the lookup table is less than that of the cut-out portion of the DFG, then the overall delay of the DFG implementation can be reduced. In the case of two or more narrow points in the DFG, the narrow point that occurs with the smallest propagation delay (i.e, closest to the "top" of the graph) will yield the most improvement in overall propagation delay.

A simple diagram of a lookup table is shown in Figure 6.3. The different input values can be treated as a single value whose important parameter is the total number of bits, n. The number of table entries is 2^n. The lookup table output is determined by the outputs being replaced in the original DFG, and again, the most important factor is the total

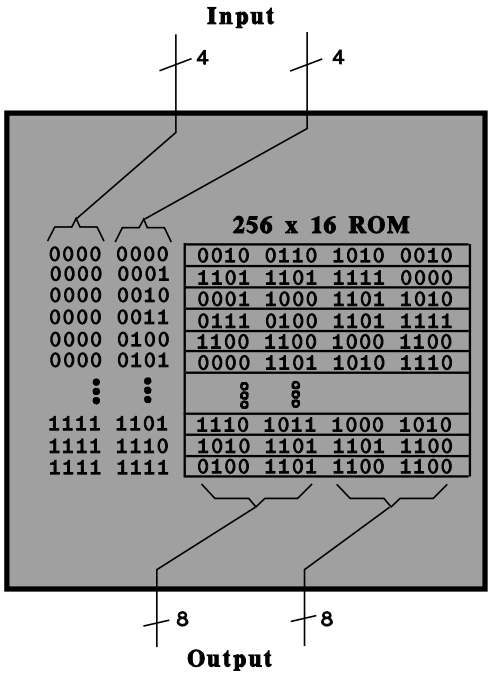


Figure 6.3: A lookup table node

number of bits. This parameter determines the size of each table entry.

Since the reconfigurable computing hardware being targeted by the SA-C compiler system is based on FPGAs, and since FPGA CLBs are based on lookup tables themselves, the implementation of lookup tables can be done efficiently. The propagation delay of a lookup table is usually small, less than all but the most trivial of combinational circuits. On the other hand, the size of a lookup table is exponentially related to the number of input bits, and linearly related to the number of output bits. The size of the lookup table, therefore, can grow very quickly, especially with the number of input bits. It appears, therefore, that this optimization will trade execution speed for the size of the overall implementation. It may not be appropriate for many DFGs. For example, a DFG that does not have a narrow point will not benefit, because the size of the required lookup table would be prohibitive. Another example is a DFG whose narrow point is near the bottom of the graph, where the substitution of a lookup table will not significantly decrease the overall propagation delay of the circuit.

6.2 Time and Space Requirements of Synthesized Lookup Tables

In order to gain some understanding of the tradeoffs that can be expected by replacing portions of a DFG with a lookup table, several experiments were run. The purpose of the experiments is to quantify the behavior of lookup tables, both in terms of the space occupied and in terms of time. These parameters should be useful in determining when the lookup table optimization might yield good results.

To perform the tests, a general-purpose "lookup table" module was written in VHDL. This module was designed so that the input and output sizes (in number of bits) is easily modified. The module itself is very simple - it simply uses a concatenation of the input bits as an index into a `constant` array. The width of each element in the array is determined by the number of the output bits. For the initial experiments, the data for the array was generated randomly. A DFG containing a "lookup table" (LUT) node was created by hand, and the DFG-to-VHDL translator was modified to handle this new node type. This modification generates a port map reference to the LUT module in the generated VHDL inner loop body, and creates a modification to the synthesis "project" (.prj) files so that the lookup table module is incorporated in the final design. The port map to the LUT module appears like those used to interface to the components in the module library that were created for some of the other DFG nodes. However, the LUT module will not come from a library; rather, it will eventually be created by a program that utilizes the "cut-out" portion of the DFG to generate the actual lookup-table values.

The VHDL generated by the modified translator was submitted to the VHDL synthesizer, Synplify, and the resulting design was analyzed to understand the effects of the lookup table modifications. First, Synplify recognized the constant lookup table array, and synthesized it as a ROM. The constant nature of the LUT enables special optimizations built into Synplify to be used in generating the FPGA circuit netlist in the implementation.

While analyzing the FPGA primitives used to implement DFGs, it was observed that the ROM arrays for the constant-value LUTs are implemented using $1x32$ ROM primitives, each of which utilizes 2 FMAPs on the chip. The $1x32$ ROMs are used as shown in Figure

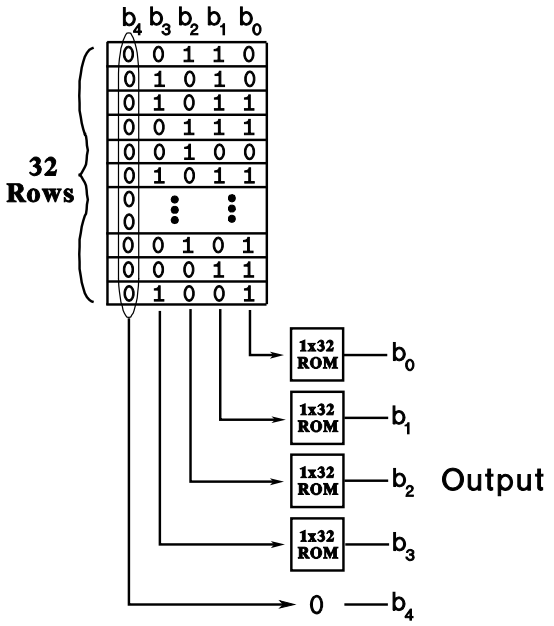


Figure 6.4: An optimized implementation of a lookup table with repeating zeroes

6.4 - one ROM is used for each "vertical" 32 bit slice within the ROM array. If the entire group of 32 bits within a slice has the same value (a `0` or a `1`), then the FMAPs that would normally be required for that column is optimized away.

This arrangement of vertical stripes is fast, since it allows the bits that make up a particular "word" within the ROM array to be accessed in parallel, one bit per 32 bit ROM. Therefore, lookup time is quite fast, and is independent of the width of the word (i.e., the size of the output value). In summary, LUT implementations on the FPGA are quite efficient in time, requiring only two CLB propagation delays for a LUT access – generally much faster than the equivalent implementation using some form of combinational logic.

LUT implementations are themselves quite predictable in terms of space, at least for "worst case" analysis. It requires one FMAP per 16 bits, except for the occasional situation where a constant column can be dissolved. This situation appears to occur infrequently when synthesizing typical LUTs, perhaps about 5% of the time. Thus, a reasonably accurate and conservative estimate of FPGA space required by a LUT can be determined by dividing the number of bits by 16, resulting in the number of FMAPs required. (FMAPs are almost always the determining resource when calculating FPGA

space.)

The comparison between the resources required by LUTs vs. the original logic is more difficult to predict, except that in nearly all cases, logic takes less space than the replacement LUT. Thus, the replacement of parts of the DFG with an LUT nearly always becomes a tradeoff of speed vs. space on the FPGA, in all but the most trivial cases.

6.3 Algorithm 1 - A Global Mincut through the DFG

To determine where the lookup table should be placed, we need to find where the mincuts are located in the DFG – that is, the places in the graph where a "horizontal" cut through the graph slices through the fewest number of bits. It is possible to find these places "by inspection" for small, simple DFGs; however, it is considerably more difficult for non-trivial codes. This fact suggests an automated process.

There may be several narrow places in the DFG. In these cases we want to select the one closest to the top of the graph, where the cut point will have the largest propagation delay relative to the bottom of the graph. In this way, the maximum amount of propagation delay can be replaced with the lookup table. The cut point does not need to be the narrowest place in the graph; it simply must be "narrow enough" so that the size of the resulting lookup table will fit on the FPGA.

An algorithm used to find the narrowest point in a graph is based on the Ford-Fulkerson algorithm (Ford and Fulkerson 1962). This algorithm finds the maximum "flow," or capacity, in a graph. This maximum flow is limited by the capacity of a set of edges in the graph called the *mincut* of the graph - the "narrowest" capacity in the graph. For a dataflow graph, if the bit-width of each edge represents its "capacity," the algorithm will determine the mincut, or narrowest point in the graph. The original algorithm has been adapted for efficient solution on a digital computer (Papadimitriou and Steiglitz 1982; Tarjan 1983).

The algorithm is described as follows: Consider a graph G, consisting of n vertices V, a starting vertex s, an ending vertex t, and m edges E, where each edge from vertex i to j has a defined flow capacity (or bit-width, in our case) $b(i, j)$. That is, $G = (s, t, V, E, b)$ with

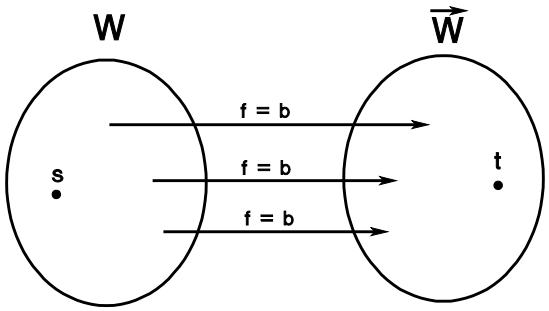


Figure 6.5: A graph partitioned into two parts, with a mincut between them

$n = |V|$ and $m = |E|$. Each edge can have a flow, denoted $f(i, j)$, such that $f(i, j) \leq b(i, j)$. The maximum capacity of the the network is reached when $f(i, j) = b(i, j)$ in each edge that constitutes the mincut; the edge is said to be *saturated*. The mincut can be identified by partitioning the graph into two parts W and $\overline{W}$, such that $s \in W$ and $t \in \overline{W}$, and where W and $\overline{W}$ are connected by arcs where $f = b$. This situation is pictured in Figure 6.5. There may be several different partitionings of a graph that satisfy these conditions, indicating that there are several mincuts for the graph.

The algorithm is iterative. Starting at s, and initially with $f = 0$ in all edges, the algorithm attempts to find a non-saturated path to t, while at the same time keeping track of the maximum amount of additional flow the path can carry. If it can find such a path to t, then it adds this maximum flow to the edges in the path, and then tries again. When the algorithm is not able to find a non-saturated flow path from s to t during an iteration, the maximum flow through the graph has been determined.

More specifically, the algorithm works by attempting to "label" each vertex in the graph. This label contains the information about the flow path being established. The label consists of two parts - the node from which the flow is coming, and the maximum flow that can occur along the path at this point. The maximum flow is determined either by the amount of flow coming from the node, or by the capacity of the edge between the nodes. In the former case, the flow has been limited by some edge previously visited; in the latter case, the flow into this node is being limited because the edge between the two nodes has become saturated. Put more concisely, if we are labeling a node j, being labeled from node i, the two part label $(L1, L2)$ for j is thus defined:

$$\begin{aligned} L1(j) &= i; \\ L2(j) &= \min(L2(i), b(i,j) - f(i,j)); \end{aligned}$$

It is also possible that a node can be labeled from one of its successor nodes – that is, a node can be labeled by traversing an edge backwards. This can happen if the edges between a node and its successor are already saturated (from a previous iteration of the algorithm). This situation has the effect of "cutting off" the node from its successor, so the node would not get labeled. To insure that these missed nodes get considered, the node is labeled as follows:

$$\begin{aligned} L1(j) &= -i; \\ L2(j) &= \min(L2(i), f(j,i)); \end{aligned}$$

In other words, the negative sign on $L1$ indicates this backwards labeling; the flow is either the current forward flow between the nodes, or a reduced flow brought from the predecessor node.

The algorithm starts by trying to label each node to which it is connected, a process called *scanning*. If a node is labeled, either from a forward or backward edge, it means that the node *could* be on the flow path being constructed – thus, we need to continue the scanning process from this node. Each labeled node is therefore put on a list; the algorithm continues the scanning process for each node on the list until either we reach t, or there are no more labeled nodes to process. If we reach the end node t, then we can use the $L1$ values to reconstruct the path we took to reach t, and we *augment* the flow along the path by the amount of flow that reached t. We then repeat the scanning process again, this time using the augmented flow values. Each pass, therefore, increases the flow along some path through the graph. Eventually, the maximum flow through the graph will be reached – this is the situation where we can no longer reach t along any unsaturated paths.

The specific algorithm, adapted from the algorithm presented in (Tarjan 1983), is presented in Figure 6.6. When the algorithm terminates, the partitions W and $\overline{W}$ can be determined by observing the labels on the nodes – if a node is labeled, it is in partition W, if not, it is in $\overline{W}$. The mincut, therefore, crosses those edges that connect labeled nodes with non-labeled ones.

```
void main()
  {
   initialize:
     for all edges, set f = 0;             // initialize all flows to zero

   again:              // Top of main iterative loop
     for all nodes i, set L1, L2 = 0;     // Reset all labels
     LIST = {s};                          // initially, do scan from s
     while ( LIST != NULL )
       {
        remove a node x from LIST;
        scan(x);                          //labelling process described below
        if ( L1(t) != 0)    // node t is labeled
          {
            for each edge from i to j along augmentation path
              {
                set f(i,j) += L2(t);   // augment flow along path
                go to again;
              }
          } // end if
       } // end while
  } // end main

void scan(x)    // labeling algorithm, scanning from node x
  {
   forward:     // forward labeling
      for each node i, a successor to node x
        {
          if (L1(i) == 0)      // node i is unlabeled
             if (f(x,i) < b(x,i)) // unsaturated, so label this node
               {
                L1(i) = x;
                L2(i) = min( L2(x), b(x, i) - f(x, i);
                Put node i on LIST;
               }
        } // end for
   backward:    // backward labeling
      for each node i, a predecessor to node x
        {
          if (L1(i) == 0)      // node i is unlabeled
             if (f(i,x) > 0)   // positive flow, so label this node
               {
                L1(i) = x;
                L2(i) = min( L2(x), b(x, i) - f(x, i);
                Put node i on LIST;
               }
        } // end for
  } // end scan
```

Figure 6.6: The Ford-Fulkerson mincut algorithm

In order to use the mincut algorithm with a DFG, a couple of modifications to the DFG were made. First, the mincut algorithm requires that the graph contain a single starting node s. A DFG, on the other hand, usually starts with several `INPUT` nodes. These nodes typically convey constant parameters to several of the other nodes in the graph, such as the generator nodes and collector nodes. They do not need to participate in the mincut calculations. Therefore they are effectively removed before the mincut analysis takes place. Once the `INPUT` nodes are removed removed, one or more of the data generator nodes become the first nodes in the DFG. In order that the graph always have a single common start node, an "s" node is created and connected to the generator nodes. The edges used to connect this start node with the rest of the DFG are defined with an "infinite" bit width (actually a very high value), so they will never have any effect on the mincut calculation.

Secondly, the mincut algorithm requires a single ending node t. In a DFG, this requirement is met, since the final node is always an `OUTPUT` node. However, the input to this node is always a single-bit "done" signal, which, since it is only one bit wide, will always be selected as the mincut! This situation can be remedied by redefining the done signal to be of infinite width – it will then never be selected, and also will not affect the rest of the mincut algorithm.

In some cases, a output of a node is connected to several input nodes. In the DFG, this is represented by several edges emanating from a single output, and connecting to several different nodes as inputs. The bit width of such an arrangement should be considered to be that of the single output, and not the sum of the widths of the individual edges. The DFG data structure was modified to insure that this multiple output situation is handled correctly by the algorithm.

Finally, the mincut algorithm as described above finds only one mincut of the graph. In fact, there may be several equal mincuts, some of which may result in a smaller overall propagation delay, depending on where they occur in the DFG. The particular mincut found by the algorithm can be manipulated by changing the order in which the labeled nodes are placed on the list to be scanned. Proper ordering of this list will cause the

algorithm to find a mincut that produces a large propagation delay, but no ordering can *guarantee* that the best mincut will always be found.

In fact, the mincut of the graph is not necessarily the most important consideration for the problem at hand. Rather, we would like to find a cut point in the DFG that is "acceptably narrow" – that is, the bit width at the cut point is small enough to produce a lookup table that will fit on the FPGA – but produces the largest possible reduction in propagation delay.

To accomplish this task, the original mincut algorithm was modified to find several successive mincuts, or near mincuts, of the graph. Once a mincut is found, an estimate of the amount of savings in propagation delay and the size of the lookup table required by the bit width of the mincut are computed. If the estimates predict that this mincut produces an acceptable reduction in propagation delay, then the algorithm terminates. If additional solutions are sought, the graph is modified by setting the bit width of one of the edges along the cut to "infinity," and the mincut algorithm is run again. Since the modified edge will no longer be part of any plausible solution, the algorithm will find a new mincut. This new result can be compared to previous ones, and the "best" solution can be chosen.

A couple of points are in order concerning this modification to the mincut algorithm. When a mincut is found, the savings in propagation delay is determined by the edge with the highest propagation delay. Thus, when selecting an edge to set to infinity, it should be the one with the highest propagation delay, since this will encourage the algorithm to find a mincut with a larger propagation delay.

6.4 Mincut Algorithm Results

The modified Ford-Fulkerson algorithm was run on several of the benchmark programs used in previous experiments. While the algorithm found narrow mincuts in some of the smaller DFGs, none of the more significant benchmarks provided mincuts that were of practical use in replacing with lookup tables – in all cases, the mincut was too "wide." Upon closer inspection, there appear to be several reasons for this:

- The mincut for all the tested algorithms includes two or more edges with bit widths of 8 or more, resulting in a lookup table requirement of 64K entries or more. This is a very common occurrence, corresponding to a node that performs a binary operation on two 8-bit values. Since such operations usually produce an 8-bit result, each of the entries in the lookup table is 8 bits wide. Therefore, the resulting LUT is 512K bits or larger in size, too large to be effectively implemented on current FPGA technologies.

- Many of the compiler optimizations that are applied before a dataflow graph is created attempt to increase parallelism – ordinarily a good thing. Unfortunately these optimizations tend to make the mincut problem even worse by increasing the "width" of the DFG. In some cases, the optimizations can be turned off, but this defeats the purpose of the optimization. The LUT solution becomes a tradeoff with, rather than additive with, the other optimizations.

- Most DFGs include control bits that extend from near the top of the graph to data collection nodes at the bottom. The edges produced by these control bits are generally narrow (generally only 1 or two bits wide); nonetheless, they aggravate the problem by contributing to both the input and output bit count of the LUT.

Nothing can be done with the bit widths of the input edges, unless the data within the algorithm can itself be narrowed. To try to mitigate the last problem, a modification was made to the algorithm, to eliminate the control lines from the mincut algorithm. Because of the method used to implement the data collector nodes, control lines are almost always connected to nodes near the bottom of the DFG, as shown in Figure 6.7a. The mincut algorithm can be modified so that it excludes these lines in the mincut calculation, as shown in Figure 6.7b.

While this modification improved the situation slightly, it did not change the fact that none of the DFGs generated for the larger algorithms contained mincuts that were narrow enough to allow effective replacement with LUTs. This method may become useful in the future, when FPGAs with even larger capacity than today's models are available.

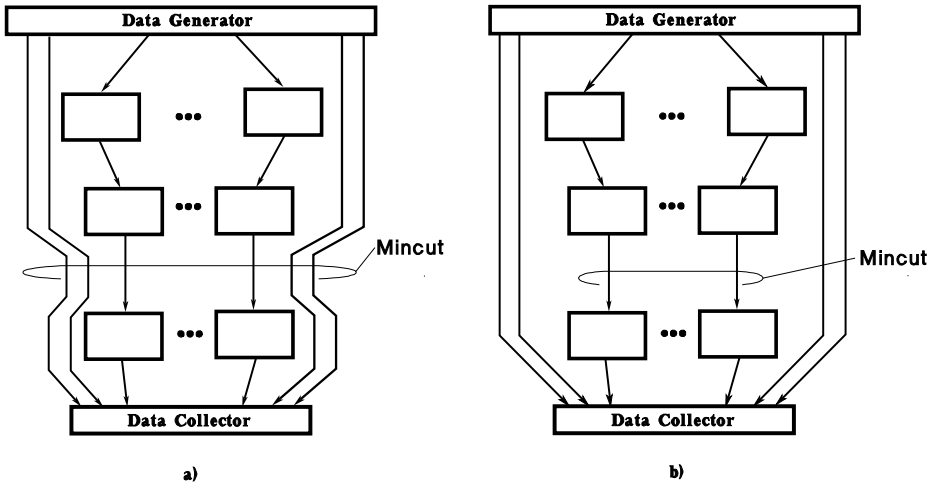


Figure 6.7: a) A mincut found within a DFG by the original mincut algorithm, and b) The modified version that eliminates control lines from the mincut

6.5 Algorithm 2 - Local Subgraphs and the "Flood" Algorithm

The method in the previous section, using the Ford-Fulkerson algorithm, takes a "global" approach to the lookup-table problem. That is, the mincut is used across the entire dataflow graph. Since most dataflow graphs are too wide to yield useful results, it may be possible to take a more "local" approach to the problem. The question becomes - are there subgraphs within a larger dataflow graph that could be replaced with lookup tables? These subgraphs would not necessarily span the entire graph, but nonetheless under the right circumstances their replacement might yield some savings in overall propagation delay. In the case where a dataflow graph has been "widened" by the compiler's attempt to increase parallelism, several of these local subgraphs may have been created; replacing each subgraph with a lookup table may produce propagation delay savings without requiring that the parallelism effects be turned off, as is the case with the first algorithm.

The concept of identifying local subgraphs within a larger dataflow graph was used in a technique called *threaded dataflow* (Papadopoulos and Traub 1991). Used on a dataflow processor called Monsoon (Papadopoulos and Culler 1990), subgraphs that represent non-parallel sequences were transformed into sequential threads that could be submitted to the processor's pipeline for more efficient execution than the original set of dataflow nodes. Here, the purpose is somewhat the same, although the details are different - the sequences

can be replaced with lookup tables so they can be executed quickly.

Following are characteristics of subgraphs that might be suitable for replacement by lookup tables:

- The subgraph starts at a node with a small number of non-constant input bits. Since the size of the lookup table is related to 2^n, where n is the number of input bits, only nodes with a relatively small number of input bits will be useful as the subgraph's starting point. Constant input bits, however, can be incorporated into the lookup table without increasing its size.

- The subgraph has a single exit point (node).

- There cannot be any inputs into the subgraph, other than through the top of the subgraph. If there are any such lines, they must be treated as input bits, and therefore add to the size of the lookup table.

- The propagation delay between top and bottom nodes of the subgraph must be larger than that for the lookup table, otherwise the replacement will not yield any savings in execution time.

A pictorial view of these requirements is shown in Figure 6.8. The DFG portions enclosed in dashed lines are candidates for consideration as subgraphs. The example on the left meets the requirements above; however, the subgraph on the right does not.

In order to test this idea, an algorithm was developed to find all the subgraphs in a DFG. Like the previous mincut algorithm, this algorithm is also a "flow" algorithm - however, it is based on a "flood" technique, rather than maximum flow. Specifically, the flood algorithm works as follows:

1. A suitable starting node is selected, called the *top node*. This node, and the possible subgraph below it, represents a candidate for replacement by a lookup table.

2. A "flood" is initiated from the top node to all the nodes connected to its outputs. That is, an approximately equal portion of the total flow is distributed to each of

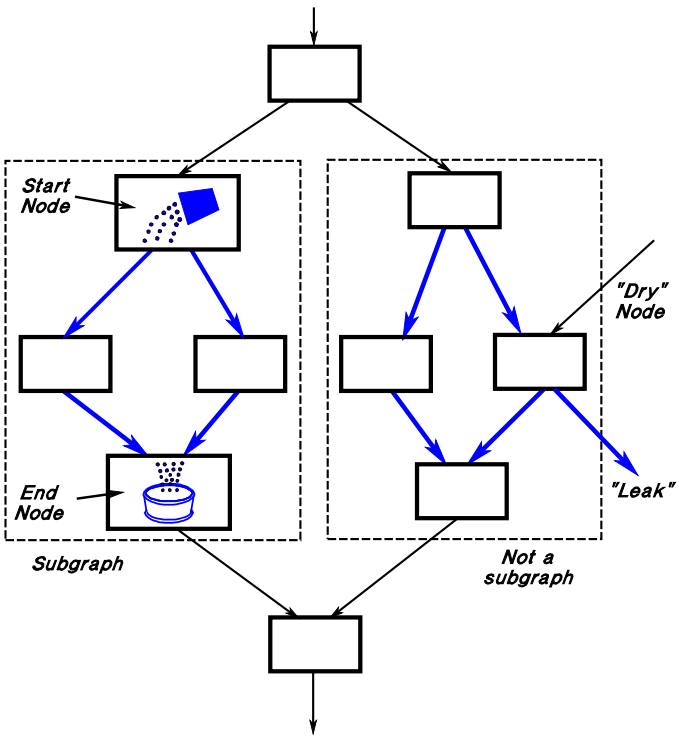


Figure 6.8: Pictorial view of two subgraph candidates. The one on the left is a subgraph; the one on the right is not.

the nodes connected below the top node, such that the entire amount of the flood is sent to the output nodes.

3. For each node connected below the top node, the amount of flow into the node's inputs is summed up and distributed to each of its outputs.

4. Step 3 is repeated recursively for each node involved in the flood. This process continues until one of the following conditions is met:

   - The flood reaches the end (bottom) of the graph, or
   - A node is found that has the entire flood on its inputs.

   In the first case, flow has been determined for at least one path from the top node, but the total flow has not yet propagated throughout all the nodes that make up the flood path, Therefore the algorithm continues by propagating flow out of those nodes that have not yet been processed. In the second case, a potential subgraph

has been found, extending from the top node through this node, called the bottom node.

5. Once the bottom node has been found, the inputs of all the nodes in the flood path are checked. If it is a subgraph, the inputs should either be connected to nodes that provide flow, or to constant values. A "dry" input (i.e., one with no flow) indicates that it is connected to a part of the DFG that is outside the bounds of the flood, indicating an external connection and therefore not a subgraph.

When a subgraph is found, the propagation delay of the subgraph can be calculated. If this delay is longer than that of a lookup table that might replace the subgraph, this subgraph is added to the list of subgraphs found within the DFG. This list represents the subgraph candidates that can be considered for replacement within the DFG.

The above algorithm finds a *minimal* subgraph - that is, the shortest subgraph. In some instances, it was observed that some subgraphs had a "tail" - a set of sequential nodes that follow the minimal subgraph. If these nodes are themselves subgraphs (i.e., no external inputs), they can be grafted onto the minimal subgraph to create a larger subgraph. This optimization has been added to the algorithm.

One advantage of the subgraph approach when compared with the first algorithm is that it is still effective even after optimizations, such as loop unrolling or stripmining, have been performed. These optimizations increase parallelism by replicating portions of the DFG, including any subgraphs. While these optimizations tend to increase the bit width of a mincut, thereby exponentially increasing the size of any lookup table that might be employed, the flood algorithm will find each replicated subgraph independently. Each replicated subgraph will require a separate lookup table, but the increase in size will be linear, rather than exponential. Thus, the subgraph approach can prove useful in even highly optimized DFG graphs, making the interaction of compiler optimizations with lookup table replacements much more likely to be additive rather than mutually exclusive.

The running time of the algorithm can be improved in some cases by graph pruning. It is not necessary to continue the flood calculations below nodes whose flow at all non-

constant inputs has not been determined as yet. This situation can occur for two reasons:

- The flood has simply not propagated to the input yet. In this case subsequent processing will eventually flood the input, at which time the algorithm can continue.
- The input is not connected to a node in the flood path. In this case the node is not part of a subgraph.

This second case introduces an additional termination situation for the algorithm - if all nodes connected to the top node have been processed, but the total flow at the end of the graph does not equal the total flow, this is not a subgraph. This optimization has been added to the algorithm; it speeds up the calculations in a substantial number of cases.

6.6 Flood Algorithm Results

Before running the flood algorithm on all the benchmarks, a couple of SA-C programs were written specifically to exercise the algorithm and observe its behavior. The first is a program that uses a subroutine to do division. The division algorithm used is the common restoring subtract-shift procedure, described in (Hwang 1979), among others. For values of a certain size (8 bits in this case), it has a fixed number of iterations, and the iterations must be performed sequentially. Thus the resulting DFG is rather "deep," and the propagation delay is rather large. There is little parallelism in the DFG. This test case was chosen since division is often implemented explicitly as a lookup table; here it is interesting to see the relationship between a manually implemented lookup table and the choices generated by the subgraph algorithm.

The second program determines if a given number is prime. Again, the most straightforward way to do this is to divide the number by all of the integers smaller than the square root of the number, and record the results. However, a couple of additional variations were used - one using a chain of *if* statements, and another using a *case* statement. The if chain translates to a sequence of comparisons, and therefore has a rather long propagation delay, while the case version translates into a parallel set of comparisons that is considerably faster. This algorithm was chosen because, like division, it can be (manually) implemented

as a lookup table.

When the subgraph algorithm is executed on these examples, not surprisingly one of the subgraphs found is the subgraph comprising the entire subroutine. The subroutine is in-lined and therefore to some extent it loses its identity as a subroutine; nonetheless the "single entry - single exit" characteristic of the original subroutine is embodied in the DFG, and since this characteristics is the definition of a subgraph, the algorithm finds it. The lookup table that could replace the subgraph is the same size as the one that could be manually generated for the subroutine, since the number of input bits (the sum of the bit widths of all the input values) and the number of output bits (the sum of the bit widths of the return values) are the same as for a manually generated lookup table. In the case of the division routine, the input width is the sum of the bit widths of the two input values, and is therefore rather large (64K for two 8 bit inputs and an 8 bit result). The prime number routine is a smaller lookup table, since the input is a single number (8 bits) and the result (i.e., output) is a single bit (signifying a "true" or "false" answer). However, in both cases the result of the algorithm is to produce a lookup table that could just as easily be manually generated. This is not a surprising result, although it does seem to validate the proper operation of the subgraph algorithm.

However, the algorithm also finds other smaller subgraphs that lie within the larger subroutine subgraph - some of these provide some more interesting (and less obvious) results. For example, the divide routine needs to determine if the AND-ing of a 16 bit value with a constant is equal to zero. This sequence, when taken as a subgraph, has 16 input bits and 1 output bit, thereby requiring 64K total bits of lookup table. The lookup table, however, requires only half of the propagation delay of the original two node DFG implementation. Since this code is inside a loop that gets unrolled eight times, the propagation delay savings also gets multiplied by a factor of 8, for a total time savings of about 21%.

These small subgraphs are analogous to "peephole optimizations" as performed in normal compiler optimizations - small localized sequences that can be replaced with more efficient code. The usual method in compiler design for detecting such optimizations is

Benchmark Name	Propagation Delay				Lookup Table			
	Orig (ns)	New (ns)	Improvement (ns)	Pct	No	in	out	Size (bits)
AddS_opt	40.2	21.1	28.2	52.5%	2	8	8	4096
Add_uint8	35.2	21.1	23.2	59.9%	1	17	8	1048576
Conv2DSumSq-strip_4_3	153.3	61	101.4	39.8%	18	16	8	9437184
Conv2DSumSq	153.3	61	101.4	39.8%	9	16	8	4718592
Conv2D-strip_4_3	126.9	61	75	48.1%	12	18	8	25165824
Conv2D-strip_5_3	126.9	61	75	48.1%	21	18	8	44040192
Conv2D	126.9	61	75	48.1%	9	18	8	18874368
ConvSep2D	229.6	163.7	75	71.3%	25	17	8	26214400
CrossCorr	82.8	39.9	52	48.2%	3	16	16	3145728
MultiplyScale	176.8	21.1	164.8	11.9%	1	16	8	524288
Multiply8	78.1	21.1	66.1	27.0%	1	16	8	524288
Square	61.4	26.1	44.4	42.5%	1	8	8	2048

Table 6.1: Results of the subgraph algorithm

to maintain a list of patterns that can be replaced with smaller sequences. However, only those patterns that have been previously identified and added to the list will be detected. An advantage of the subgraph approach is that the sequences need not be identified ahead of time - the algorithm will detect all subgraphs.

Table 6.1 shows the results of executing the subgraph algorithm on several SA-C programs. The table lists the total number of subgraphs found in each DFG, along with the original propagation delay and potential new delay if all the subgraphs are replaced with LUTs. The percentage improvement in propagation delay is also shown, along with an estimate of the number and size of LUTs that would be required for this improved propagation delay.

In interpreting the results shown in table 6.1, it is clear that significant speed improvements can be attained in some of the benchmarks by replacing subgraphs with LUTs, but at considerable, and in most cases prohibitive, cost in space. In most of the examples, the space required for LUTs exceeds the capacity of even very large FPGAs available today. However, this data demonstrates the potential of LUTs in reducing propagation delay; as FPGA densities continue to increase, these large space requirements may become more feasible.

Some observations are in order as a result of analyzing the results from the flood

algorithm. Many of the subgraphs identified by the algorithm contain multiply (`IMUL` or `UMUL`) nodes. These nodes are quite expensive in terms of propagation delay. This is not a very surprising result – lookup tables have been used since the earliest days of computing to implement multiplication, so the flood algorithm confirms the conventional wisdom. However, the flood algorithm sometimes finds a larger subgraph that includes a multiply operation combined with other nodes. In this case, the entire subgraph can be replaced with a LUT, resulting in better overall performance than would be obtained by just replacing the multiply node individually.

Subgraphs that have inputs that are constants are especially good candidates for replacement. Since the constant value can be "built into" the LUT, the size (number of bits) used by the constant doesn't contribute to the total input bit count, resulting in a much smaller LUT. In terms of generating more suitable subgraphs for replacement, it might be appropriate for the compiler to attempt to produce more constant input values. For example, several subgraphs found in the benchmark programs have one or more inputs directly connected to an `INPUT` node; the `INPUT` node represents a variable that is usually defined from an input file, or is a parameter value derived from the structure of the loop generators. This value is not a constant by traditional definition, but its value *is* constant throughout the execution of a particular input set. A design change in the compiler runtime system might allow the subgraph to be replaced with a LUT specifically generated for the dataset.

In conclusion, the subgraph algorithm appears to be useful in some cases, including long strings of sequential nodes, expensive (in terms of propagation delay) nodes and sequences, and smaller sequences that have a small number of inputs and/or outputs. The speedup improvement can be significant, although the increased cost in terms of space on the FPGA can also be significant. It would appear that the subgraph replacement technique is not particularly useful, at least for image processing applications, which dominate the SA-C benchmarks that were used. It may be that other types of algorithms may respond better to the technique. With the continuing trend of more dense FPGA chips, some with dedicated onboard memory, this optimization might become more attractive in the future.

Chapter 7

Conclusion and Future Work

Reconfigurable computing systems can achieve very impressive performance when compared to conventional processors for many types of image processing and other parallel applications. Up until now, the programming methods for RCS's have hampered their widespread deployment - the skills required have been those of hardware designers, rather than application programmers. However, with the development of the SA-C language, this situation has changed.

This dissertation has described the design and implementation of the "back end" of the SA-C compilation system. It accepts dataflow graphs produced by the SA-C compiler, and translates them into VHDL which can be processed by commercial synthesis tools to produce a binary executable that can be downloaded and executed on the RCS. The compiler, coupled with this back end, comprise a system that can produce high performance RCS solutions in a single command, and without the need for the specific hardware knowledge usually required for using an RCS.

This dissertation first describes a manual, non automated implementation of a common image processing algorithm - the Prewitt edge detection algorithm. This exercise helped to expose many of the issues associated with programming an RCS, and provided a solution that could be used as a benchmark against which automated solutions could be compared.

Next, the design of the DFG-to-VHDL translator is described. Using the DFG as input, this translator generates a VHDL description of the inner loop body, created from scratch. It then determines the types of data generator and data collector modules that must be included in the final design. Finally, it generates a "glue" module that ties the

other modules together into a single system. The entire set of VHDL modules can then be submitted to synthesis and place-and-route tools to produce a binary executable. This executable can be downloaded to the RCS and executed under the control of a program produced by the compiler and running on the host processor. Experimental results show that the design produced by the automated process is competitive in performance with the manual design. Furthermore, the automated design can be realized in a matter of hours, whereas the manual process took several weeks to complete.

Following the completion of a functional DFG-to-VHDL translator, the dissertation next describes the development of some optimizations that can be performed on the DFG prior to the translator operation. The first optimization attempts to reduce the total propagation delay through the inner loop body by using pipelining. The first step in this process requires knowledge of the propagation delays through each node in the DFG. Since the exact values of these delays are not known until after the synthesis and place-and-route steps, which can take hours to complete, an estimator tool has been created. The estimator was developed by collecting actual timing data from numerous synthesis runs involving each DFG node, and then developing a series of equations to fit the data. Tests on the resulting estimate equations show that the estimator produces results that are usually within 10% or so of actual.

Using the estimator, pipeline registers are placed in the ILB at the proper places, and the timing between the data generator and data collector is adjusted to account for the pipeline stage delay. The pipeline registers have the effect of splitting the ILB into two or more smaller pieces, thereby allowing the RCS to run at a faster overall clock speed.

The final optimization involves the replacement of portions of the DFG with lookup tables. LUTs are very time-efficient when synthesized on FPGA-based RCS's - they can replace complicated nodes and node sequences in the DFG, thereby increasing the clock speed, albeit at the expense of FPGA space. Two different algorithms were developed to identify sub-graphs within the DFG. The first one, based on the Ford-Fulkerson maximal flow algorithm, identifies a "mincut" through the DFG - a point at which the DFG can be split into top and bottom parts. The bottom portion is then replaced with a LUT.

This method was able to improve performance in some cases, but the resulting LUTs tended to be too large to be practical using today's technology. A second algorithm identified all of the subgraphs within the DFG, then produced a list ranked according to the greatest potential for performance improvements. The second algorithm proved to be more versatile than the first. It identified two specific types of subgraphs - first, rather small subgraphs that could be replaced with relatively small lookup tables, that also produced relatively modest performance increases, and secondly, larger subgraphs that could produce significant speed improvements, but that were quite large. It appears that LUT replacement should be used selectively, since the greatest performance gains also require the largest amount of space.

In the future, some improvements and enhancements are anticipated for the SA-C compilation system. The timing estimation model, used for pipelining, can be improved by refining the regression methods, and hence the model equations used. In a similar project, that improved a similar space estimation model, could be applied here. In that project, the accuracy of space estimates for DFGs were improved from around 10% to less than 5% for most problems. An improved timing estimator would tend to improve the accuracy of pipeline register placement.

While the SA-C compilation system was specifically developed to target a specific RCS system, the AMS Wildforce processor, it was nonetheless designed to be easy to port to other RCS systems. Some experience with the porting process has been gained while adapting the current system to target the next generation Starfire processor. An effort is beginning to re-target the SA-C compilation system to the FPPA processor, described earlier in Section 2.3.4(Rinker, Nathan, Buehler, and McConaghy 2005). Because of careful partitioning of the components into hardware independent and hardware dependent parts, a relatively small amount of the system will need to be rewritten when targeting a different RCS.

It is believed that at the time it first became fully operational, the complete SA-C compilation system, including the SA-C compiler, developed as part of the Cameron project (outside of this dissertation), and the DFG-to-VHDL translator as described in this disser-

tation, is the first fully automated system able to produce RCS programs, without human intervention, using a single command. The resulting designs appear to be competitive with manually generate ones, even before any optimizations are applied. Both the SA-C compiler and the DFG-to-VHDL translator provide significant optimizations that can be used to improve the performance - some of these can be selectively activated so the application programmer can control the tradeoffs used in developing the final design. In many cases, the optimizations that can be applied and evaluated in a matter of minutes using the automated process, would require a complete redesign if attempted manually. This ability significantly advances the state-of-the-art in RCS programming capability.

REFERENCES

The Cameron Project. Information about the Cameron Project, including several publications, is available at the project's web site, `www.cs.colostate.edu/cameron`.

Agarwal, A., S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, and M. Srikrishna, D.and Taylor (1997, August). The RAW compiler project. In *Proc. Second SUIF Compiler Workshop*.

Annapolis Micro Systems, Inc., Annapolis, MD (1997). *WILDFORCE Reference Manual*. Annapolis Micro Systems, Inc., Annapolis, MD. `www.annapmicro.com`.

Annapolis Micro Systems, Inc., Annapolis, MD (1999a). *STARFIRE Reference Manual*. Annapolis Micro Systems, Inc., Annapolis, MD. `www.annapmicro.com`.

Annapolis Micro Systems, Inc., Annapolis, MD (1999b). *WILDSTAR Reference Manual*. Annapolis Micro Systems, Inc., Annapolis, MD. `www.annapmicro.com`.

Arvind and R. Iannucci (1983). A critique of multiprocessing von Neumann style. In *Int. Symp. on Computer Architecture*. Stockholm, Sweden.

Athanas, P. M. and A. L. Abbott (1994). Processing images in real time on a custom computing platform. In R. W. Hartenstein and M. Z. Servit (Eds.), *Field-Programmable Logic Architectures, Synthesis, and Applications*, pp. 156–167. Springer-Verlag, Berlin.

Banerjee, P., N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden (1999, August). MATCH: a MATLAB compiler for configurable computing systems. Technical Report CPDC-TR-9908-013, Center for Parallel and distributed Computing, Northwestern University.

Barua, R., W. Lee, S. Amarasinghe, and A. Agarwal (2001, November). Compiler support for scalable and efficient memory systems. *IEEE Transactions on Computers*.

Blinn, J. F. (1993, July). What's the deal with the DCT? *IEEE Computer Graphics and Applications 13*(4), 78–83.

Böhm, A., R. Beveridge, B. Draper, C. Ross, M. Chawathe, and W. Najjar (2002, April). Compiling ATR probing codes for execution on FPGA hardware. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa Valley, CA.

Böhm, A., B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross (2001, April). One-step compilation on image processing applications to FPGAs. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, Rohnert Park, CA.

Böhm, A., J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar (2002). Mapping a single assignment programming language to reconfigurable systems. *Supercomputing 21*, 117–130.

Buehler, D. M. (2004). *A Methodology for Designing and Analyzing Fixed-Point Implementations of Computational Data Paths*. Ph. D. thesis, University of Idaho.

Buehler, D. M., G. W. Donohoe, and P. S. Yeh (2004, September). A software tool for designing fixed-point implementations of computational data paths for embedded and reconfigurable computational environments. In *Proc. 7th Annual Intl. Conf. on Military and Aerospace Programmable Logic Devices (MAPLD)*.

Buell, D., J. Arnold, and W. Kleinfelder (1996). *Splash 2: FPGAs in a Custom Computing Machine*. IEEE CS Press.

Consortium, V. (1997, October). Vector signal image processing library forum. `www.vsipl.org`.

DeHon, A. (1998, Nov.). Comparing computing machines: Technology and applications. In *Proceedings of SPIE 3526*.

DeHon, A. and J. Wawrzynek (1997). The case for reconfigurable processors.

DeHon, A. and J. Wawrzynek (1999, June). Reconfigurable computing: What, why, and implications for design automation. In *Proceedings of Design Automation Conference (DAC '99)*, pp. 610–615.

Dennis, J. and D. Misunas (1975, January). A preliminary architecture for a basic data-flow processor. In *Proc. 2nd Intl. Symp. on Computer Architecture (ISCA)*, pp. 126–132.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik (1), 269 – 271.

Donohoe, G. W., D. M. Buehler, and S. Bruder (2000a, November). A design strategy for fixed word length data paths. In *Proc. 8th Annual NASA Symp. on VLSI Design*.

Donohoe, G. W., D. M. Buehler, and S. Bruder (2000b). A reconfigurable data path processor for space applications. In *Proc. Military and Aerospace Programmable Logic Devices (MAPLD)*.

Draper, B., R. Beveridge, A. Böhm, C. Ross, and M. Chawathe (2002, August). Implementing image applications on FPGAs. In *Intl. Conf on Pattern Recognition*, Quebec City.

Draper, B., W. Najjar, A. Böhm, J. Hammes, R. Rinker, C. Ross, and J. Bins (2000, Sept 11-13). Compiling and optimizing image processing algorithms for FPGAs. In *IEEE Intl. Workshop on Computer Architecture for Machine Perception (CAMP)*, Padova, Italy.

Estrin, G. (1963, December). Parallel processing in a restructurable computer system. In *IEEE Trans. on Electronic Computers*, pp. 747–755.

Ford, L. R. and D. R. Fulkerson (1962). *Flows in Networks*. Princeton, NJ: Princeton University Press.

Goldstein, S. C. and M. Budiu (1999). *The DIL Language and Compiler Manual*. Carnegie Mellon University. `www.ece.cmu.edu/research/piperench/dil.ps`.

Goldstein, S. C. and H. Schmit (1999, October). Instant hardware: Fast compilation to scalable reconfigurable hardware. Presentation made to DARPA ACS-PI Meeting, San Juan, Puerto Rico.

Goldstein, S. C., H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer (1999). Piperench: A coprocessor for streaming multimedia acceleration. In *Proc. Intl. Symp. on Computer Architecture (ISCA '99)*. `www.cs.cmu.edu/~mihaib/research/isca99.ps.gz`.

Gonzalez, R. C. and R. E. Woods (1992). *Digital Image Processing*. Reading, MA: Addison-Wesley.

Group, O. H. C. (1997). The Handel language. Technical report, Oxford University.

Hammes, J. (2000, April). *Compiling SA-C to Reconfigurable Computing Systems*. Ph. D. thesis, Department of Computer Science, Colorado State University.

Hammes, J., A. Böhm, C. Ross, M. Chawathe, B. Draper, R. Rinker, , and W. Najjar (2001, April). Loop fusion and temporal common subexpression elimination in window-based loops. In *IPDPS 8th Reconfigurable Architectures Workshop*, San Francisco.

Hammes, J. and W. Böhm (1999). *The SA-C Language - Version 1.0*. Colorado State University. Document available from `www.cs.colostate.edu/cameron`.

Hammes, J., B. Draper, and W. Böhm (1999, January). Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems. In *Proceedings: International Conference on Vision Systems*, Las Palmas de Gran Canaria, Spain, pp. 522–537.

Hammes, J., R. Rinker, A. Böhm, and W. Najjar (1999, October). Compiling a high-level language to reconfigurable systems. In *Compiler and Architecture Support for Embedded Systems (CASES)*, Washington, DC.

Hammes, J., R. Rinker, W. Böhm, W. Najjar, B. Draper, and J. R. Beveridge (1999, July). Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Denver, CO.

Hammes, J., R. Rinker, W. Najjar, and B. Draper (2000, June 26-29). A high-level algorithmic programming language and compiler for reconfigurable systems. In *The 2nd Intl Workshop on the Engineering of Reconfigurable Hardware/Software Objects (ENREGLE), part of 2000 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, NV.

Hammes, J., R. E. Rinker, D. M. McClure, A. P. W. Böhm, and W. A. Najjar (2001). *The SA-C Compiler Dataflow Description*. Colorado State University. Document available from `www.cs.colostate.edu/cameron`.

Hauck, S. (1998, June). The future of reconfigurable systems. Keynote Address, 5th Canadian Conference on Field Programmable Devices, Montreal.

Hauser, J. R. and J. Wawrzynek (1997). Garp: A MIPS processor with a reconfigurable coprocessor. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 12–21. IEEE. `www.cs.berkeley.edu/projects/brass/documents/GarpProcessors.html`.

Hennessey, J. L. and D. L. Patterson (2003). *Computer Architecture: A Quantitative Approach* (Third ed.). San Francisco: Morgan Kaufmann Publishers, Inc.

Hoang, D. (1993). Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185–192. CS Press, Loa Alamitos, CA.

Hwang, K. (1979). *Computer Arithmetic: Principles, Architecture, and Design*. New York: John Wiley and Sons.

Hwang, K. and F. A. Briggs (1984). *Computer Architecture and Parallel Processing*. New York: McGraw-Hill Book Company.

Jones, M., L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott (1997, Apr.). Implementing an API of distributed adaptive computing systems. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*.

Konstantinides, K. and J. Rasure (1994, May). The Khoros software development environment for image and signal processing. In *IEEE Transactions on Image Processing*, Volume 3, pp. 243–252.

Kulkarni, D. (2001, June). Configurable resource estimation for lookup table based FPGAs. Master's thesis, University of California, Riverside.

Kulkarni, D., W. A. Najjar, R. E. Rinker, and F. J. Kurhadi. Compile-time area estimation for LUT-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems*. Accepted for publication.

Lee, W., R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe (1998, October). Space-time scheduling of instruction-level parallelism on a RAW machine. In *Proc. Eighth Intl. Conf. on Architectural Support for Programming languages and Operating Systems ASPLOS-VIII*.

Loeffler, C., A. Ligtenberg, and G. Moschytz (1989). Practical fast 1-D DCT algorithms with 11 multiplications. In *Internation Conference on Accoustics, Speech and Signal Processing*, pp. 988–992.

Lu, G., H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi (1999, Sept.). The Morphosis parallel reconfigurable system. In *Proc. of EuroPar 99*.

Mangione-Smith, W. (1997a, Oct.). Application design for configurable computing. *IEEE Computer 30*, 115–117.

Mangione-Smith, W. (1997b, Dec.). Seeking solutions in configurable computing. *IEEE Computer 30*, 38–43.

Najjar, W., B. Draper, W. Böhm, and J. R. Beveridge (1998, Oct.). The Cameron project: High-level programming of image processing applications on reconfigurable computing machines. In *PACT '98 - Workshop on Reconfigurable Computing*, Paris, pp. 83–88.

Najjar, W. and J. Worley (1997). Optimized compilation of visual programs for image processing on adaptive computing systems. Research proposal to DARPA, Colorado State University and Khoral Research, Inc.

Najjar, W. A., A. Böhm, B. Draper, J. Hammes, R. Rinker, R. Beveridge, M. Chawathe, and C. Ross (2003, August). High-level language abstraction for reconfigurable computing. *IEEE Computer 36*(8), 63–69.

Natarajan, S., B. Levine, C. Tan, D. Newport, and D. Bouldin (1999). Automatic mapping of Khoros-based applications to adaptive computing systems. Technical report, University of Tennessee. Available from `http://microsys6.engr.utk.edu:80/~bouldin/darpa/mapld2/mapld_paper.pdf`.

Papadimitriou, C. H. and K. Steiglitz (1982). Maximum flow. In *Combinatorial Optimization: Algorithms and Complexity*, Chapter 27, pp. 579 – 629. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Papadopoulos, G. and D. Culler (1990, June). Monsoon: An explicit token-store architecture. In *Proc. 17th Intl. Symp. on Computer Architecture (ISCA)*, pp. 82–91.

Papadopoulos, G. and K. Traub (1991, May). Multithreading: A revisionist view of dataflow architectures. In *Proc. 18th Intl. Symp. on Computer Architecture (ISCA)*, pp. 342–351.

Patel, A. (2000, June). Design of library modules for automated synthesis of dataflow graphs. Master's thesis, Department of Computer Science, Colorado State University.

Patterson, D. L. and J. L. Hennessy (1998). *Computer Organization & Design: The Hardware/Software Interface* (Second ed.). San Francisco: Morgan Kaufmann Publishers, Inc.

Peleg, A., S. Wilkie, and U. Weiser (1997, January). Intel MMX for multimedia PCs. *Communications of the ACM 40*(1), 24–38.

Periyayacheri, S., A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee (1999, November). Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB. In *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conf. (PDCS'99)*.

Prewitt, J. M. S. (1970). Object enhancement and extraction. In B. S. Lipkin and A. Rosenfeld (Eds.), *Picture Processing and Psychopictorics*. Academic Press, New York.

Pryor, D. V., M. R. Thistle, and N. Shirazi (1993). Text searching on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 172–178. CS Press, Loa Alamitos, CA.

Ratha, N. K. and A. K. Jain (1999, Jan.). Computer vision algorithms on reconfigurable logic arrays. In *IEEE Trans. on Parallel and Distributed Systems*, Volume 10-1, pp. 29–43.

Ratha, N. K., D. T. Jain, and D. T. Rover (1995). Convolution on Splash 2. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 204–213. CS Press, Loa Alamitos, CA.

Ratha, N. K., D. T. Jain, and D. T. Rover (1996). Fingerprint matching on Splash 2. In *Splash 2: FPGAs in a Custom Computing Machine*, pp. 117–140. IEEE CS Press.

Rinker, R., M. Carter, A. Patel, M. Chawathe, and W. Najjar. Compiling a high level language into reconfigurable hardware. Available from `www.cs.colostate.edu/cameron`.

Rinker, R., M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and A. Böhm (2001, February). An automated process for compiling dataflow graphs

into reconfigurable hardware. In *IEEE Transactions on VLSI Design*, Volume 9, pp. 130–139.

Rinker, R., J. Hammes, W. Najjar, A. Böhm, and B. Draper (2000, July 10-12). Compiling image processing applications to reconfigurable hardware. In *IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, Boston, MA.

Rinker, R., P. Nathan, D. Buehler, and A. McConaghy (2005, October). Using dataflow graphs to automate the programming of a reconfigurable processor. In *Proc. 12th Annual NASA Symp. on VLSI Design*.

Rose, J., A. El Gamal, and A. Sangiovanni-Vincentelli (1993). Architecture of field-programmable gate arrays. *Proceedings of the IEEE 81*(7), 1013–1029.

Sabde, J., D. M. Buehler, and G. W. Donohoe (2003). Focal plane array sensor readout correction on a reconfigurable processor. In *Proc. 11th Annual NASA Symp. on VLSI Design*.

Sanchez, E., M. Sipper, J. Haenni, J. Beuchat, A. Stauffer, and A. Perez-Uribe (1999, June). Static and dynamic configurable systems. *IEEE Trans. on Computers 48*(6).

Schmit, H., D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor (2002). Piperench: A virtualized programmable datapath in 0.18 micron technology. In *IEEE Custom Integrated Circuits Conference (CICC)*.

Schott, B., S. Crago, C. C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti. Reconfigurable architectures for systems level applications of adaptive computing. Available from `http://www.east.isi.edu/SLAAC/`.

Silc, J., B. Robic, and T. Ungerer (1998, March). Asynchrony in parallel computing: from dataflow to multithreading. *Parallel and Distributed Computing Practices 1*(1).

Synplicity, Inc. (1999). *Synplify User's Guide, Release 5.2.2*. Synplicity, Inc. `www.synplicity.com`.

Tarjan, R. E. (1983). Primal-dual algorithms for max-flow and shortest path: Ford-fulkerson and dijkstra. In *Data Structures and Network Algorithms*, Chapter 6, pp. 117 – 136. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Taylor, M. (2004, September). The raw prototype design document. Technical report, Massachusetts Institute of Technology.

Umbaugh, S. E. (1998). *Computer Vision and Image Processing: a practical approach using CVIPtools*. London: Prentice-Hall.

Waingold, E., M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal (1997, September). Baring it all to software: RAW machines. *Computer*.

Wallace, G. K. (1991). The JPEG still picture compression standard. *Communications of the ACM 34*(4), 30–44.

Xilinx, Inc. (1999, Oct.). *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*. Xilinx, Inc. `www.xilinx.com`.

Xilinx, Inc., San Jose, CA. (1998). *The Programmable Logic Databook*. Xilinx, Inc., San Jose, CA. `www.xilinx.com`.