Architecture-Independent Design for Run-Time Reconfigurable Custom Computing Machines

Rhett Daniel Hudson

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Electrical Engineering

Peter M. Athanas, Chairperson
Scott F. Midkiff
Walling R. Cyre
Mark T. Jones
James A. Fabunmi

May 2000
Bradley Department of Electrical
And Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, Configurable Computing, Run-Time Reconfigurable, Automated Design

Architecture-Independent Design for Run-Time Reconfigurable Custom Computing Machines

Rhett D. Hudson

(ABSTRACT)

The configurable computing research community has provided a wealth of evidence that computational platforms based on FPGA technology are capable of cost-effectively accelerating certain kinds of computations. One actively growing area in the research community examines the benefits to computation that can be gained by reconfiguring the FPGAs in a system during the execution of an application. This technique is commonly referred to as run-time reconfiguration. Widespread acceptance of run-time reconfigurable custom computing depends upon the existence of high-level automated design tools. Given the wide variety of available platforms and the rate that the technology is evolving, a set of architecturally independent tools that provide the ability to port applications between different architectures will allow application-based intellectual property to be easily migrated between platforms. A Java implementation of such a toolset, called Janus, is presented and analyzed here. In this environment, developers create a Java class that describes the structural behavior of an application. The design framework allows hardware and software modules to be freely intermixed. During the compilation phase of the development process, the Janus tools analyze the structure of the application and adapt it to the target architecture. Janus is capable of structuring the run-time behavior of an application to take advantage of the resources available on the platform. Examples of applications developed using the toolset are presented. The performance of the applications is reported. The retargeting of applications for multiple hardware architectures is demonstrated.

Dedication

For the Brothers of Alpha Phi Omega, in Leadership, Friendship and Service…

Acknowledgements

There are many people to thank. I'd like to thank Ben Crawford, who told me I should really finish. Jim Peterson, for being an excellent friend and for all his mathematical pointers. Ray Bittner for playing the Woodrow Call to my Augustus McCrae. Lou Pochet, for doing all those things that only Lou can do. Wendy Akers for always being willing to listening to me. The AEDAR Corporation for giving me a taste of why the work was worth it.

I'd like to thank Annapolis Micro Systems for knowing that it's not all about having the diploma in your hand.

I'd like to thank the members of the original Janus project team. Jason Hess, James Atwell, David Moye, Ken Shiring and Zhimei Ding. I forgot one…

David Lehn. I could never have finished this degree without Dave's help. He Java expertise was invaluable. His GUI's were flashy. And we all know, it doesn't matter what the application does or how well it works. If the GUI is flashy, that's a quality piece of software.

I'd like to thank my parents. They were encouraging throughout the process and listened to a lot of complaining and ranting.

And as far as complaining and ranting goes, I have to offer a special thanks to all of my roommates over the years who put up with what became generally known as "core-breaches".

Thanks go out the members of my committee who provided advice and support throughout the years. Especially Dr. Midkiff and Dr. Athanas.

Finally, this work is dedicated to the brothers of Alpha Phi Omega. They gave me the best four years of my life, so far, and taught me so many things about how to live that I could not list them here if I used all the pages between these covers.

Table of Contents

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 5

2.1 Representative RTR CCM Applications 5
2.1.1 RTR Applications with a Run-Time Nature 5
2.1.2 RTR Applications with Virtual Hardware 7
2.1.3 RTR Applications with Dynamically Instantiated Hardware 8

2.2 Automated Design Tools 10
2.2.1 Automated Design of Non-RTR Applications 10
2.2.2 Automated Design of RTR Applications 12
2.2.3 Structural Design of non-RTR CCM Applications 13
2.2.4 Just Another Hardware Description Language 14

2.3 Dynamic Allocation of Hardware Resources 21

2.4 Janus Design Goals 22
2.4.1 Unified RTR Application Description 22
2.4.2 Time-Multiplexed Virtual Hardware 22
2.4.3 A Pragmatic Approach 23
2.4.4 Architecture Independence 23

CHAPTER 3 JANUS 24

3.1 Application Development 27
3.1.1 Creating the Application Model 27
3.1.2 Creating the User Interface 31
3.1.3 Compiling the Application 31

3.2 Application Execution 35

CHAPTER 4 ANATOMY OF AN APPLICATION: IMAGE INTERPOLATION 38

4.1 Image Interpolation 38

4.2 Approach 39
4.2.1 Computational Structure of Image Interpolation 40
4.2.2 Temporal Partitioning 41

4.3 The Application Interface 41

4.4 Operations 43

4.5 Building the Hardware with JHDL 48

CHAPTER 5 ARCHITECTURALLY RETARGETABLE COMPILATION 53

5.1 The Hardware Abstraction 53

5.2 The `RunTime` Class 58

5.3 The `Compiler` Class 60

CHAPTER 6 APPLICATION DEVELOPMENT 63

6.1 Design Flow 63
6.1.1 Initial Design Partitioning 63
6.1.2 Software Implementation 63
6.1.3 Incremental Hardware Development 64
6.1.4 Application Compilation 65
6.1.5 Application Execution 66

6.2 Image Interpolator 67

6.3 Stereo Depth Finding 68

6.4 Wavelet Compression 69

CHAPTER 7 RESULTS 70

7.1 Goals Revisited 70
7.1.1 Unified Design 70
7.1.2 Virtual Hardware 71
7.1.3 Pragmatism 72
7.1.4 Hardware Independence 80

7.2 Limitations and Opportunities 81
7.2.1 Operator Memory Width 81
7.2.2 Operator Size 82
7.2.3 Operator Types 83
7.2.4 Operator Ordering 83
7.2.5 Dynamic Operations 84

7.3 Future Directions 84
7.3.1 Tower of Power Backend 84
7.3.2 Wavelet Compression Application 85
7.3.3 Simulation of Multiple Memory Port Backends 85
7.3.4 And Beyond… 85

REFERENCES 86

Table of Figures

Figure 2-1: The Just Another Browser Window. 15
Figure 2-2: JAB Schematic Views. 16
Figure 2-3: WILDFORCE Simulation Environment. 17
Figure 2-4: Schematic for Absolute Value Operator. 19
Figure 3-1: Use Case Diagram for the Janus Toolset. 26
Figure 3-2: Package Diagram for Janus Abstraction. 26
Figure 3-3: Extended Application Development Use Case. 27
Figure 3-4: Class Diagram of the Stages Package. 29
Figure 3-5: Object Diagram of Image Interpolator. 29
Figure 3-6: Janus Hardware Abstraction. 32
Figure 3-7: Concrete WILDFORCE Model. 32
Figure 3-8: Interaction of Classes during Compilation. 33
Figure 3-9: Sample Hardware Configuration. 34
Figure 3-10: Activity Diagram for Compute Cycle. 36
Figure 3-11: Basic Scheduling Algorithm. 36
Figure 3-12: Application Execution Sequence Diagram. 37
Figure 4-1: The Inverse Filer [FerP97]. 39
Figure 4-2: The Fast Spline Transform [FerP97]. 39
Figure 4-3: Computational Structure of Image Interpolation. 40
Figure 4-4: Generalized Filter Tap. 50
Figure 4-5 : Possible Filter Taps. 52
Figure 6-1: Janus Control Center. 66
Figure 6-2: Graphical User Interface for the Image Interpolator. 67
Figure 6-3: Graphical User Interface for Stereo Depth Finder. 68
Figure 6-4: Wavelet Compression Decomposition Layer. 69
Figure 6-5: Three Layer Wavelet Compression Transform. 69
Figure 7-1: Interpolation 64x64 1 PE. 74
Figure 7-2: Interpolation 128x128 1 PE. 74
Figure 7-3: Interpolation 256x256 1 PE. 74
Figure 7-4: Interpolation 256x256 1 PE. 74
Figure 7-5: Run Times with Varying Input Sizes. 75
Figure 7-6: Configuration Overhead with Varying Input Sizes. 76
Figure 7-7: Extrapolated Configuration Overhead for Multiple Frames. 76
Figure 7-8: Run-Time Characteristics with Multiple PEs. 77
Figure 7-9: Hardware Computation Time with Multiple PEs. 77
Figure 7-10: Stereo Depth Finding 1 PE. 78
Figure 7-11: Wavelet Compression in 1 PE. 79
Figure 7-12: Configuration Overhead with Respect to Computational Time. 80

Table of Code Listings

Listing 2-1: Sample of JDHL Structural Logic. 19
Listing 4-1 : The Application Interface. 41
Listing 4-2: The Image Interpolator's build() Method. 42
Listing 4-3 : Operation Interface. 44
Listing 4-4: InvRowOp's userToCcm() Method. 45
Listing 4-5: InvRowOp's ccmToBuffer(). 45
Listing 4-6: InvColOp's userToCcm() Method. 46
Listing 4-7: InvColOps's ccmToBuffer() and bufferToUser() Methods. 46
Listing 4-8: Code Fragment from the Run-Time System. 47
Listing 4-9 : Core JHDL Design for a Filter. 49
Listing 4-10 : Core JHDL Design for a Filter Tap. 51
Listing 4-11 : Tap's getTap() Method. 52
Listing 5-1 : `HardwareArchitecture` Pseudo-Class Definition. 54
Listing 5-2: `ProcessingElement` Pseudo-Class Definition. 54
Listing 5-3: `WildForceXL_4062` Constructor. 55
Listing 5-4: Abbreviated `W4_XC4062` Class Definition. 55
Listing 5-5: `Configuration` Pseudo-Class Definition. 57
Listing 5-6: `ConfigureProcessingElements` Code Fragment. 58
Listing 5-7: `FetchMemory` Code Fragment. 59
Listing 5-8: `RetireMemory` Code Fragment. 60
Listing 5-9: Interlude Event Code Fragment. 60

Chapter 1 Introduction

A large body of configurable computing research has dealt with the ability to rapidly prototype application specific hardware using field programmable gate arrays (FPGAs). Typically, these applications are developed for a fixed array of FPGAs and supporting components organized in a custom-computing machine (CCM). The development of applications for these CCMs utilizes design techniques inherited from the application specific integrated circuit (ASIC) community. Typical commercial FPGA design tools are essentially ASIC design tools that have been retargeted from VLSI cell libraries to support libraries of primitives for FPGAs. Since these applications are based on ASIC design techniques, the design tools assume that the FPGAs will be configured once at startup and remain in the same configuration throughout the execution of an application. One obvious limitation of this approach is that there are always computations that require more resources than are available on a fixed array of FPGAs.

While ASIC solutions must adopt this kind of static design, applications based on the inherently flexible FPGA do not. The computing engines that take advantage of this flexibility are referred to as run-time reconfigurable (RTR) CCMs. RTR applications solve the scalability problem of traditional rapid prototyping techniques by adopting a divide and conquer approach. Large problems are broken down into separate computations, each of which fits onto the array. The first computation performs computations and stores the results into memory. The array is then reconfigured for the next computation, which computes results based on the results of previous computations. This process continues until all the computations have been performed [HudL98][EldH94]. Using these techniques, the primary limitation on the size of problem that an RTR application can solve is the size of the memory required to store intermediate results [EldH96].

Application research using non-RTR CCMs has demonstrated a wealth of applications that can be accelerated by taking advantage of the highly-parallel computing environment available within an array of FPGAs. While the computational advantage of the technology is evident, several difficulties have kept the general-purpose computing market from adopting the technology. One of the difficulties is that high-level tools to aid development of applications are not generally available. The tools that do exist are largely targeted toward specific architectures.

The objective of this dissertation is to demonstrate and evaluate a set of application development tools for the RTR CCM environment that make the design flow simpler.

These tools, called Janus[1], make the following contributions to RTR CCM application design flow:

- A demonstration, using real applications, of a unified design framework for developing RTR CCM applications,
- An object-oriented approach that allows developers to build applications without knowledge of the architecture, compilation methodology or run-time implementation of the target CCM,
- Examples of how designs can be automatically adapted to take advantage of the features available within a target architecture,
- A stage-based design paradigm that provides an intuitive approach to designing RTR applications,
- An evaluation of the successes and shortcomings of the approach.

The current state of the art in RTR application design is an ad hoc process. No commercial tools exist that support the RTR design flow. RTR developers currently must maintain a delicate balance between several hardware designs, software control of the CCM platform, and run-time interaction between the hardware and the software. The Janus tools present a unified design flow that allows the hardware, software and interaction between them to be completely described using the high-level language Java [GosJ96]. Computational hardware is specified using Brigham Young University's JHDL, a structural hardware description language built within the Java language [HutB99, BelH98]. The software component of an application is built within the Janus framework using the Java language itself and any of its ancillary programming interfaces. Software interaction with hardware is provided automatically by the Janus tools. Interaction with the CCM vendor's application programming interface (API) is performed through the Java Native Interface (JNI) [GorR98].

Since the architecture of RTR CCM platforms can vary widely from platform to platform, the tools provide an architecturally independent view of RTR CCM technology to the application developer. Using the Janus framework, application developers can work without detailed knowledge of the target architecture. The number of FPGAs, the number of memory ports per FPGA, the amount of logic available on an FPGA, and the native application-programming interface for run-time control of the hardware are all handled by the development tools instead of the developer. The developer can recompile an application to any target architecture supported by the system. This provides the ability to port designs between different CCM architectures. Furthermore, since the life cycle of configurable computing platforms is relatively short and the development cycle for custom software tools for a platform is relatively long, a useful design framework must support architectures that have not yet been developed.

[1] Janus is the Roman god of doors and good beginnings. He is commonly portrayed as having two faces, one looking toward the past and one looking toward the future [HamE42]. Hopefully, the Janus tools are a door leading toward a new form of computation. They are, at least, a good beginning.

The primary advantage of a CCM is that it is configurable. A microprocessor has a limited number of specific computational units. It may have one multiplier, one adder and a barrel shifter. If an application never uses the barrel shifter, the silicon that implements the barrel shifter goes to waste. In a CCM, if an application needs ten adders, the CCM can be configured to provide ten adders. If an application needs four multipliers, the FPGAs can be configured to provide four multipliers.

This configurability allows developers to exploit parallelism in an application. Advanced super-scalar microprocessors may be able to execute several instructions simultaneously under ideal circumstances. CCMs can easily execute the equivalent of hundreds of instructions simultaneously. Typically, these instructions can be pipelined to provide even greater parallelism. Microprocessors spend approximately 30% of their time performing compares and branch instructions associated with program control structures [FelR94]. CCMs can integrate control directly into the structures that are implemented on the FPGAs. This allows them to remove control overhead from the computational stream.

Further, Princeton microprocessor architectures are limited to a single memory interface through which microprocessors fetch both their data and their instructions. CCM architectures do not have to perform instruction fetches to drive their computation. Harvard microprocessor architectures have an additional memory interface for fetching instructions, but are still limited to a single interface for data. CCMs are typically designed with at least one memory interface per FPGA and architectures that are more recent have moved to supporting multiple memory interfaces per FPGA.

These features of CCMs provide the means to gain the significant performance increases demonstrated in the literature. Any successful development paradigm for RTR CCMs must allow applications to exploit these underlying properties of the target architecture. The Janus tools present an abstraction of the underlying CCM that allows applications to be automatically adapted to take advantage of the resources available on a target platform. The developer creates sets of operations, called stages, that need to be performed in a particular order and the tools automatically figure out how to schedule parallel instances of them on the target hardware.

The remainder of this dissertation is structured as follows. Chapter 2 examines some of the RTR applications reported in the literature and examines their common characteristics. An examination of these common characteristics induces a set of goals for an automated set of RTR design tools. Existing tools reported in the literature are evaluated with respect to the presented design goals. Brigham Young University's JHDL is examined in detail to demonstrate its suitability as a basis for fulfilling the design goals. Finally, each of the design goals is explicitly defined and the motivations for each are presented.

Chapter 3 presents a detailed examination of the Janus application development abstraction. A detailed discussion of the object-oriented structure of the tools is presented. The algorithms used by the tools are described. Chapter 4 presents the implementation of an image interpolation

application as a case study. Chapter 5 examines the details of implementing the Janus tools themselves. Chapter 6 examines the design flow of the tools at a high level. The advantages to the development process are discussed. Chapter 7 evaluates the Janus tools with respect to the design goals proposed in Chapter 2, draws some conclusions about the success of the tools, points out some shortcomings, and discusses the future of the Janus tool set.

Chapter 2 Background

Automated application development tools are motivated by the development experience of humans who have designed systems. When early computer programmers wrote enough code in an assembly language, they realized that some portions of the task could by automated using compilers and interpreters. Compilers and their associated high-level languages are based on the lessons learned from writing programs in assembler code. The same is true for development tools targeting run-time reconfigurable applications. The need for the Janus tools is derived from the nature of RTR applications reported in the literature and the inability of existing tools to address the needs of RTR application developers. This chapter examines a variety of reported applications and identifies several design representation requirements that are unique to RTR applications. With a clear understanding of the kind of designs that need to be supported, some existing development tools are examined. There are no existing tools reported in the literature that provide the required features, but existing tools do provide some lessons about how such a tool set should be designed. Some high-level-language-to-hardware compilers are examined and their advantages and disadvantages are discussed. Several structural design environments are presented and their merits with respect to the high-level language compilers are examined. Based on the arguments presented, one of the structural representations is chosen as a basis for building the Janus tools.

2.1 Representative RTR CCM Applications

Many RTR applications have been presented in the literature. This section presents a representative set of applications and examines the implications each has for the design of a general purpose set of development tools. The conclusion is that run-time reconfigurable applications have a unique set of requirements that extend beyond the boundaries of traditional design tools.

2.1.1 RTR Applications with a Run-Time Nature

Lemoine reported one of the earliest examples of a run-time reconfigurable application [LemM95]. The application was developed to accelerate searches of genomic databases. Genetic data consists of long sequences of base pairs. A basic genomic search involves finding base pair sequences in a database that match a particular pattern that the user is interested in. Lemoine created a hardware accelerator for conducting these kinds of searches by creating a long pipeline

of comparison units. Each comparison unit is designed to detect a particular base pair. Sequences of these comparators are strung together in the pipeline to look for a particular sequence of base pairs corresponding to the pattern the user wants to find. The results from all of the comparisons in the pipeline are gathered together and if enough of the comparisons are positive, a search hit is generated. The particular genetic pattern that the user is looking for, however, is only known at run time and a specialized pipeline cannot be completely built beforehand. This motivated the use of run-time reconfiguration to create a specific comparison pipeline in response to a user request. A general framework for the search engine is constant between searches, but the pipelined comparators themselves are dynamically assembled and inserted into the FPGA configuration at run-time. Each time a new search is conducted a new configuration is generated and downloaded onto the device.

Gunther presents a similar application that performs text searches [GunM96]. Users of the search system provide a set of reference documents relevant to the text that they are seeking as input to the search engine. The search engine analyzes the reference documents and develops a set of relevant keywords to search for in a text database. These text-searching algorithms function best when the keywords are coded into special purpose hardware. However, since the keywords are derived from the user supplied reference documents, the basis of the search is not known until run time. This precludes directly incorporating the keywords into the search hardware when it is compiled. This problem is identical to the problem that Lemoine encountered in the genomic searching hardware. Gunther's adopts essentially the same solution presented by Lemoine. The RTR CCM array is configured with a skeleton circuit that provides all of the basic I/O interfaces and communication busses for the application. Then, once the user has provided a set of keywords, a special purpose synthesis routine generates a hardware configuration that encodes the keywords and interconnects with the skeleton circuit. Gunther's target CCM is based on a partially reconfigurable Algotronix FPGA [Algo92]. The Algotronix chip is specially designed to allow portions of the FPGA to be reconfigured at run time. Gunther takes advantage of this by leaving the skeleton circuit in place and only generating a partial configuration that corresponds to the text-searching units. This partial configuration is then downloaded onto the computational array and is used to perform the full-text search.

Consider the representational needs of these relatively simple applications. The application must be able to query the user for a search sequence. The sequence must be converted into a pipeline of specialized comparison units. The resulting hardware must be downloaded onto the hardware platform. A database must be accessed to retrieve raw genetic data to search. That data must be streamed through the hardware pipeline. Results from the search must be retrieved from the hardware and presented to the user. Many existing tools provide the means to accomplish portions of these objectives. The Java programming language is capable enough to present the user with an interface and to query a database for data. VHDL, Verilog and schematic capture are all capable of representing the hardware design. CCM vendors provide libraries in standard programming languages for configuring the platform and moving data back and forth. None of these tools, however, provides a unified means of describing all of these behaviors in a single coherent design. None of them is capable of describing an application whose fundamental hardware components are designed in response to a run-time event.

The current state of the art requires the application developer to build all of the components of the design separately. The general search framework and the individual pipeline stages are built using a traditional ASIC-style design tool. The user interface is designed using the author's procedural programming language of choice. The user interface controls the CCM hardware using a proprietary application-programming interface supplied by the vendor. The developer manages all of the interfaces between the components. Since there are no standard design representations for these interfaces, the resulting interfaces are created ad hoc on a per application basis[2].

2.1.2 RTR Applications with Virtual Hardware

Lemoine and Gunther's RTR applications take a step beyond simple ASIC prototyping-style applications by introducing the run-time nature of RTR applications. Their hardware changes in response to demands placed on it by the user at run time. They are simple RTR applications, however, in that they only reconfigure their hardware on a search-by-search basis. Neither application performs any reconfiguration during the execution of a single search.

The automatic target recognition (ATR) system created by Villasenor [VilS96] demonstrates an RTR system that performs FPGA reconfiguration during its computational cycle. The primary computation in the ATR application is the correlation of a set of target templates over a sample image. The image data is preprocessed and reduced to a binary image. The target templates themselves are also a set of binary images. In its simplest form, the correlation computation corresponds to performing a pixel-wise logical AND operation between the template and some sub-window of the radar image and counting the number of matching 1's pixels [VilS96]. The 1's counting portion of the correlation can be accomplished with an adder tree that sums the outputs of the AND operations. The binary target templates tend to have many more 0 elements than 1 elements. Since only the 1 elements in the templates correspond to possible matches between the images, an adder tree can be tailored for each template and save a significant quantity of configurable resources. Further, many of the templates are similar to each other and share a large number of common 1 elements. Similar templates implemented on the same FPGA can share portions of their adder trees. The irregularity of the adder trees, and the potential for sharing of logic between them, creates a packing problem in terms of assigning individual templates to particular FPGAs in the CCM. The number of templates that must be compared to the radar image is much larger than the number of templates that can be mapped to a typical CCM. To solve this problem, the platform is reconfigured at run-time. Several different configurations are developed for each FPGA in the system.

The ATR application demonstrates the use of virtual hardware to increase the breadth of a computation. Lemoine and Gunther's applications are configured only once per search and the available configurable resources inherently limit the size of the search, i.e. only so many keyword or base-pair comparison units will fit on the CCM. The ATR application does not perform dynamic generation of configurations at run-time, but it does utilize run-time

[2] Recently, attempts have been made to increase the portability of host code for configurable computing platforms. One such proposal is the standard interface for distributed adaptive computing systems presented in [JonS99].

reconfiguration to increase the size of the problem that the CCM can address. Instead of building a general purpose ATR engine, Villasenor takes the approach of building several template-specific ATR engines and time-multiplexing them on the hardware. If the latency of the overall computation is unimportant, this technique allows the ATR engine to process an arbitrary number of target templates.

Where Villasenor's ATR application utilizes RTR techniques to increase the number of templates that can be processed, thus increasing the breadth of its computation, Hudson's image interpolator application utilizes RTR to increase the depth of the computation [HudL96]. Hudson's image interpolation application[3] takes, as input, a single-channel intensity image. A four-stage filtering process is used to increase its resolution by a factor of four. The target CCM for this application lacked the resources necessary to implement the entire computation. The data dependencies of the process require that each of the four stages be executed sequentially. A CCM configuration was developed to implement each of these stages. At run time, each stage is loaded onto the CCM sequentially. An input image is fed into the first stage and intermediate data between each stage is saved in memory during reconfiguration. After the fourth stage has executed, the output image is retrieved from the CCM's memory.

The distinguishing characteristic between extending the depth of a computation and extending its breadth is the requirement that intermediate data be saved between reconfigurations. The ATR application produces output data after executing each configuration. The image interpolation application produces output data only after all of its reconfigurations have been completed.

Neither of these applications generates hardware designs at run-time. All of the hardware configurations for both applications can be generated without respect to the behavior of the end user. However, both applications still exhibit a run-time nature that cannot be described by conventional design tools. ASIC-style tools are capable of producing the hardware designs. Standard programming languages are capable of sequencing the configurations on a target CCM using a proprietary vendor library. No representation, however, encapsulates both.

2.1.3 RTR Applications with Dynamically Instantiated Hardware

A microprocessor with tightly coupled RTR support is presented by Clark [ClaH96]. The Dynamic Instruction Set Computer (DISC) is a specially designed microprocessor with an attached configurable computing array. Programmers of the DISC can create custom instructions that are implemented on the configurable processing array. These specialized instructions are swapped in and out of hardware as required to execute the instruction stream going through the processor. A complete computing system is reported that allows programmers to write programs in C and seamlessly call their specialized instructions without having to reference any proprietary CCM programming interface.

The specialized hardware operations are swapped onto the configurable array as required by the execution of the microprocessor's instruction stream. The microprocessor contains a full set of

[3] The automated design of the image interpolator application is used throughout this dissertation as a specific example of the design tools presented. A much more detailed description of the engine is presented in Chapter 4.

branching instructions, so it easy to create an application whose configurable resource requirements are not known at compile time. The DISC maintains a hardware cache containing all of the most recently used instructions. If execution requires an operation that is not currently configured, the DISC performs a partial reconfiguration of the computational array to make it available. If the array does not have enough configurable resources remaining to hold the needed instruction, one or more of the least recently used instructions is replaced with the new instruction. Since the dynamic requirements for operators cannot be known either when the microprocessor's machine code is assembled or when the specialized instructions themselves are designed, the DISC must also be capable of dynamically placing new instructions on the array.

The DISC achieves this dynamic placement by requiring the hardware instructions to conform to a strict logical and physical interface specification. The logical interface insures that the microprocessor is capable of interacting with all of the specialized instructions in a consistent fashion. The physical interface insures that the hardware instructions can be dynamically placed within the computing array with minimal routing and placement overhead.

DISC-based applications represent still another new class of RTR application. They are similar to the two search engines in that the dynamic hardware cache is surrounded by static skeleton circuitry that provides interface functions between the specialized instructions and the microprocessor. DISC applications are similar to the ATR and image interpolation applications in that they are capable of time multiplexing their configurable resources to increase the amount of virtual hardware that is available. DISC applications distinguish themselves in that the sequence of configurations loaded onto the CCM is not known when the application is compiled. Further, given the hardware caching policy, the placement of the specialized instructions on the underlying FPGAs must be determined at run time.

Most recently, Patterson has presented a high-performance DES encryption algorithm that utilizes run-time reconfiguration and partial reconfiguration [PatC00]. In general, during the use of an encryption or decryption algorithm, the specific code key is not known until run time. Portions of the DES algorithm are dependent on a key sequence. This dependency is the source of the encryption algorithm's security. Patterson's hardware implementation does not explicitly take this key as an input. Instead, the key is used at run time to dynamically generate portions of the hardware configuration. Some portions of the hardware design are static with respect to the key. Therefore, Patterson's implementation can take advantage of the Virtex FPGA's partial reconfiguration capabilities to change between encryption mode and decryption mode and to change keys without performing complete reconfigurations of the device. Patterson's DES implementation fits into the same general pattern as the other RTR applications. A static skeleton is specialized at run-time to match permutations of the problem that are unknown until a user supplies a specific context.

Together, these applications present a spectrum of the design problems that RTR applications developers must face and therefore the design problems that automated tools should address. First, RTR design tools must allow the application developer to specify their design in a single coherent representation. This representation must encapsulate the design of the hardware on the

CCM accelerator, the design of the user interface on the host, and the interface between the two. A complete set of tools must provide the designer with the ability to create new hardware configurations in response to user events. The designer must be able to specify and sequence multiple hardware configurations either to increase the breadth or depth of a computation. Finally, a complete set of design tools must be able to support applications whose sequence of reconfiguration is dependent upon run-time data.

2.2 Automated Design Tools

Having identified the needs of the RTR application developer, the next step in the design of an automated tool is to determine the scope and breadth of an initial version of such a tool. This section examines some of the high-level design tools reported for design of non-RTR CCM applications and the very limited research that that has been done for the automated design of RTR applications. Based on the experiences documented in the literature, goals for the first version of the Janus tools are presented.

2.2.1 Automated Design of Non-RTR Applications

One often-adopted goal of automated tool designers is to try to raise the accessibility of CCM computational technology by relieving the application designer of the burden of designing hardware. Generally, these design tools select a high-level input language, preferably one that is familiar to a large audience, and compile that language to hardware.

Wenban presents a development system that compiles a high-level language called Hardware Promela into a hardware configuration for an FPGA accelerator board [WenB96]. The system is composed of a compiler, a debugger, a linker and an interface library. The compiler translates Hardware Promela into a netlist suitable for implementation by vendor tools. Wenban argues that non-engineers have difficulty using conventional hardware simulation software since it operates on compiled netlists rather than high-level code created by the user. In lieu of a conventional simulator, the Hardware Promela environment provides a debugger. The debugger presents a high-level interpretation of simulated circuit behavior that allows the user to watch the behavior of their circuit in the same way that a C programmer watches the execution of code in a conventional debugger. When a program is ready for execution on hardware, the linker connects the user's circuit with the hardware specific interface libraries. Code running on the host machine is written separately in C++ and interfaces with those same libraries.

The authors present the system as a development environment for embedded FPGA applications. They present two example systems, a data acquisition board and a token-ring network node. Communication between the host and the configurable boards is performed through a limited interface and is primarily a conduit for controlling and configuring the remote boards. Both applications realize relatively simple communications protocols in a real-time environment. While Wenban's results are successful within the context of his presented goals, no results are presented that demonstrate that Hardware Promela is capable of efficiently handling computations that are more complex. Its ability to extract parallel structures in its computations, for instance, is not evident. In addition, designers are fundamentally limited to producing designs for a single FPGA.

The design compiler presented by Isshiki provides a C++ facility for hardware description [IssW96]. Isshiki's compiler creates dataflow representations of the user's computations and provides for spatial partitioning of designs across multiple FPGAs. C++'s object-oriented design facilities are used to create a set of basic data types that support integer and floating-point operations. These classes have overloaded mathematical operators that allow the user to describe computations in a manner very similar to one in which a user would program procedurally in C. Instead of performing actual operations, however, the calls to the class' math operators generate a dataflow description of the computation. This dataflow graph is then passed on to a spatial partitioner that creates configurations for individual FPGAs. Sequential tasks are possible using instances of another special type that can be used as the control variables in for-loops. These variables are instantiated as counters in the synthesized hardware and can be used in expressions. Isshiki's compiler provides some steps forward in terms of allowing users to express problems that exceed the boundaries of a single FPGA and in terms providing automated extraction of parallelism. It is lacking, however, in that it supports only a very limited subset of the C language for use by the designer. In addition, automated spatial partitioning of designs for FPGAs is an NP-complete problem whose heuristic solutions are extremely sensitive to the topology of the computational dataflow graph and the target CCM [VijG90].

The development framework reported by Peterson provides a design path that begins with ANSI-C and results in a set of configurations for a multi-FPGA system [PetO96]. The compiler processes ANSI C and provides output in an intermediate dataflow representation called graph description language. The target architecture is represented abstractly so that the tools can be retargeted to multiple platforms. Given the resources available on the target architecture, the compiler creates a design from a set of library operators that fits within the target architecture and minimizes the overall run time. Operators on the computational platform can be reused and have multiple implementations with varying latency and configurable hardware requirements. The resulting partitioning and scheduling problem is addressed using simulated annealing.

Peterson's work provides unrestricted ANSI-C as its input language. It also introduces the idea of designing for an abstracted hardware target using abstracted mathematical operators. Its primary limitation is that potential applications are limited by the statically available hardware on the target architecture. Lack of large-scale computational applications implemented with the system, precludes judgment on its success as a design system.

The University of Tennessee presents a design environment called CHAMPION [NatL99]. It uses the Khoros Cantata visual programming software as its design front end. Users are provided with a set of image processing primitives that can be composed to perform image-processing operations. The CHAMPION software provides automatic retiming to match paths through the dataflow and performs multi-FPGA design partitioning. It is unclear how well the CHAMPION system abstracts the underlying hardware system and allows the user to function without detailed knowledge of the architecture. The system does provide an automatic partitioner, that allows the user to function without knowledge of the architecture, but no data has been presented on the tool's ability to map arbitrary designs to varying hardware architectures.

All of these high-level language compilers achieve, with varying success, their goal of abstracting FPGA design up to the level of traditional programming languages. Of the three, only Hardware Promela does so without introducing an NP-complete problem into the design process. The addition of the spatial partitioning algorithm and the simulated annealing process in the respective C compilers allows them to handle a much wider class of application designs. However, tool design attempts conducted as part of the exploratory work for this dissertation [HudL98], has led the design of the Janus tools away from attempts to solve NP-complete problems. This prior research indicated that an attempt to create a set of RTR design tools whose input specification was a high-level language would result in the need to solve multiple NP-complete problems. While such a set of tools might very well be possible, the initial effort indicated that a greater understanding of the overall design process was required before such a task should even be attempted. The initial goal of the Janus tool set is to provide a useful, complete design environment that avoids solving any NP-complete problems. Based on the algorithms for high-level language translation presented in the literature, avoidance of NP-complete problems precludes the adoption of a high-level language as the input specification to the Janus tools.

2.2.2 Automated Design of RTR Applications

Luk and Shirazi propose a method for automating the extraction of RTR elements from conventional design representations [LukS96,LukS97,ShiL98]. Luk presents a schematic model of reconfigurable hardware that surrounds configurable portions of a design by control blocks that behave, intuitively, like multiplexors. The model allows several different implementations of the configurable portion of the design. The different implementations of the hardware between the two control blocks are themselves represented by circuit blocks. The configurable circuit's top-level inputs are passed through the input control block and are routed to the active configuration. The output control block routes the output from the current configuration back into the rest of the circuit. Actual implementations of the model are flexible with respect to the available resources. If there are enough hardware resources on the target device, each of the configurable circuit blocks could be implemented and the control blocks could be actual multiplexors. In this case, there would be no need for run-time reconfiguration. If sufficient resources are not available, the control blocks become abstract entities in the system control unit that detect the need to change configurations and arrange for partial reconfiguration of the devices.

Luk also derives some rules for manipulating the abstraction. The rules provide a mechanism for manipulating configurable blocks and optimizing them for partial reconfiguration. Given this model of reconfiguration, the authors propose a synthesis procedure in [LukS97]. The first step of this procedure involves the decomposition of a user's design into configurable regions [ShiL98]. This process begins by taking two successive adjacent circuit configurations converting the circuit representations into dataflow graphs. The next task is to identify portions of the data flow graph that are identical and do not require reconfiguration and portions that will require reconfiguration. This is a matching problem between the nodes of the two dataflow graphs. The quality of a match between nodes is evaluated based on the types of the operations

being represented, the position of the operation in a relationally placed block or absolutely placed block, and its connectivity with other blocks in the system. The authors then apply a heuristic matching algorithm that produces a best match between the two graphs. Nodes that match exactly represent portions of the design that do not require reconfiguration. Nodes that match partially or not at all represent regions that will require reconfiguration. The algorithm represents these two configurations using the control-block model described in [LukS96].

Once these configurable regions are identified, the authors describe a tool for computing partial reconfigurations. The *ConfigDiff* tool, described in [LukS97], performs this function. Given two configurations, it computes the minimum partial reconfiguration between them. *ConfigDiff* targets the Xilinx 6200 family and supports the wildcarding facilities of those chips to minimize the partial reconfiguration.

These tools support low-level design automation for RTR applications. They specifically target partially configurable devices. The tools function best when successive designs contain many similar elements and there is a significant potential for reconfiguration overhead saving to be gained from reusing previously configured circuitry. The usefulness of the model for designs with radically dissimilar stages or for designs targeted to devices that do not support partial reconfiguration is unclear. No results have been presented that indicate the success of the algorithm in handling large applications and some portions of the algorithm are yet to be implemented.

2.2.3 Structural Design of non-RTR CCM Applications

Given that high-level languages and schematics are inappropriate for the design goals of the Janus tools, some descriptive format must be chosen as an input language. Having chosen not to use a high-level language to describe the behavior of Janus designs, the tools instead utilize a high-level language to describe the structure of the design. As has been described earlier, RTR applications have a natural bifurcation between the specification of the host behavior and the design of the configurable hardware. As will be seen later, choosing an unmodified existing full-featured programming language as a means for describing the structure of an RTR application not only unifies the description of these two facets of a design, it allows them to interact.

Several different object-oriented design tools utilizing standard procedural programming languages have been proposed as tools for structural descriptions of configurable hardware. Recent examples include Chu [ChuW98], Mencer [MenM98] and Bellows [BelH98]. All three of these use standard compilers for popular computing languages to represent structural hardware designs. Chu and Bellows both use Java, while Mencer uses C++.

Of the three, Bellows and Chu present the most well designed object representation for digital hardware. The object abstraction presented Mencer appears to be heavily dependent upon a particular CCM. The object abstraction and supporting tools do not appear to be as well developed as the ones presented by Bellows and Chu.

The Janus tools are heavily dependent upon JHDL, which is the object abstraction presented by Bellows. The choice of abstractions is based on the close ties between Virginia Tech and Brigham Young University. While the descriptions appearing in [BelH98] and [MenM98] might lead a designer to choose the [MenM98] abstraction over the JHDL abstraction, given some of its automated design aids, practical experience with the JHDL tools have proven it to be a fully functional abstraction.

2.2.4 Just Another Hardware Description Language

The Just another Hardware Description Language (JHDL) [BYU99,BelH98] is an object-oriented hardware abstraction that allows hardware developers to create FPGA hardware with an unmodified Java development system. Java was chosen because it is an intuitive, easy to learn object-oriented language [GosJ96]. The Java language is designed to be cross-platform compatible wherever compilers have been ported. The language also provides a meta-language interface, in its reflection package, that allows Java code to examine its own class interfaces and structure.

The goals of JHDL are to provide a unified way for hardware developers to describe the structure of digital logic circuits, guide placement of the circuits on actual FPGAs, and execute the design on the target platform. JHDL allows users to describe circuit construction at the same level of abstraction as purely structural VHDL. In addition, the JHDL developer has all of the constructs available to a high-level software programmer at their disposal for creating structural models. This provides a great deal more flexibility than VHDL's rather clumsy `if-generate` and `for-generate` syntax. The abstraction used to create logic primitives is also independent of the target device. This allows designs to be portable between FPGAs. Flexible support for guiding the placement of designs is also available using methods in the `Logic` class. It is difficult to specify placement in VHDL, even with VHDL 93's new `attribute` syntax, and certainly not as flexible as the mechanism available in JHDL. Finally, while it is technically possible to build host-side CCM code using VHDL, the mechanisms for implementing native extensions to VHDL are not widely used and are generally proprietary to particular VHDL vendors. The Java language has a standardized interface for native code called the Java Native Interface (JNI) [GorR98]. This allows Java programmers to provide portable interfaces to CCM platforms.

2.2.4.1 JHDL Run-Time Environment

The JHDL runtime environment centers on the Just Another Browser (JAB) window. The basic interface is pictured in Figure 2-1. JHDL hardware designs are encapsulated within test bench classes that provide the necessary simulator interfaces. When a test bench class is loaded into the JAB, the user can examine the hardware elements of the design that is loaded using the hierarchy viewer in the left pane. Information about the signals, types, widths and current values, are available in the right pane.

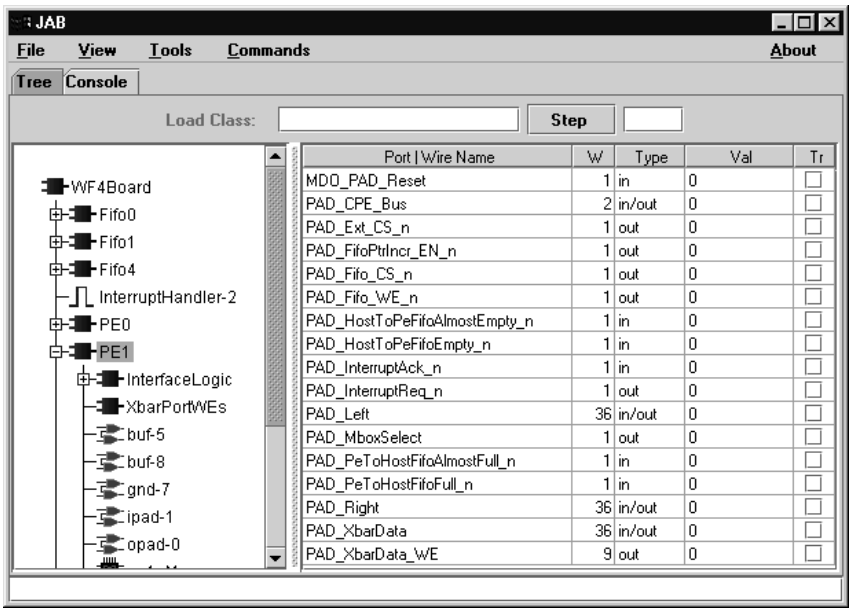

Figure 2-1: The Just Another Browser Window.

JHDL does not provide a schematic capture interface at this point in its development, but it does provide a facility for visualizing designs in schematic form. The schematic form is automatically extracted from JHDL's internal representation of the circuit and its layout is formatted using various schematic generation algorithms. Figure 2-2 shows two schematic views generated by the JAB. The schematic hierarchy can be navigated. The bottom screenshot is an internal view of the `EdgeDetector` block in the upper screenshot.

JHDL also provides a simulation environment that is tailored to FPGA development. The JHDL simulation environment allows users to perform software only simulations of their structural hardware, but with the press of a button, the simulator can run the same application test bench on the hardware platform itself. Readback capabilities of target architectures allow the simulation tools to query the values of non-boundary signals within the design and map them to the user's simulation displays. This allows the user to perform comparisons between the simulated values and actual values of all the signals in the system. Figure 2-3 shows the WILDFORCE simulation environment. Within the simulation environment, configurations can be loaded onto the various PEs. Memory can be initialized and interrogated during simulation. Schematics of the designs can be called up with the current values of the simulation or the actual values read back from the hardware displayed on their wires. The environment also supplies a built-in waveform viewer that allows users to select signals for visualization.

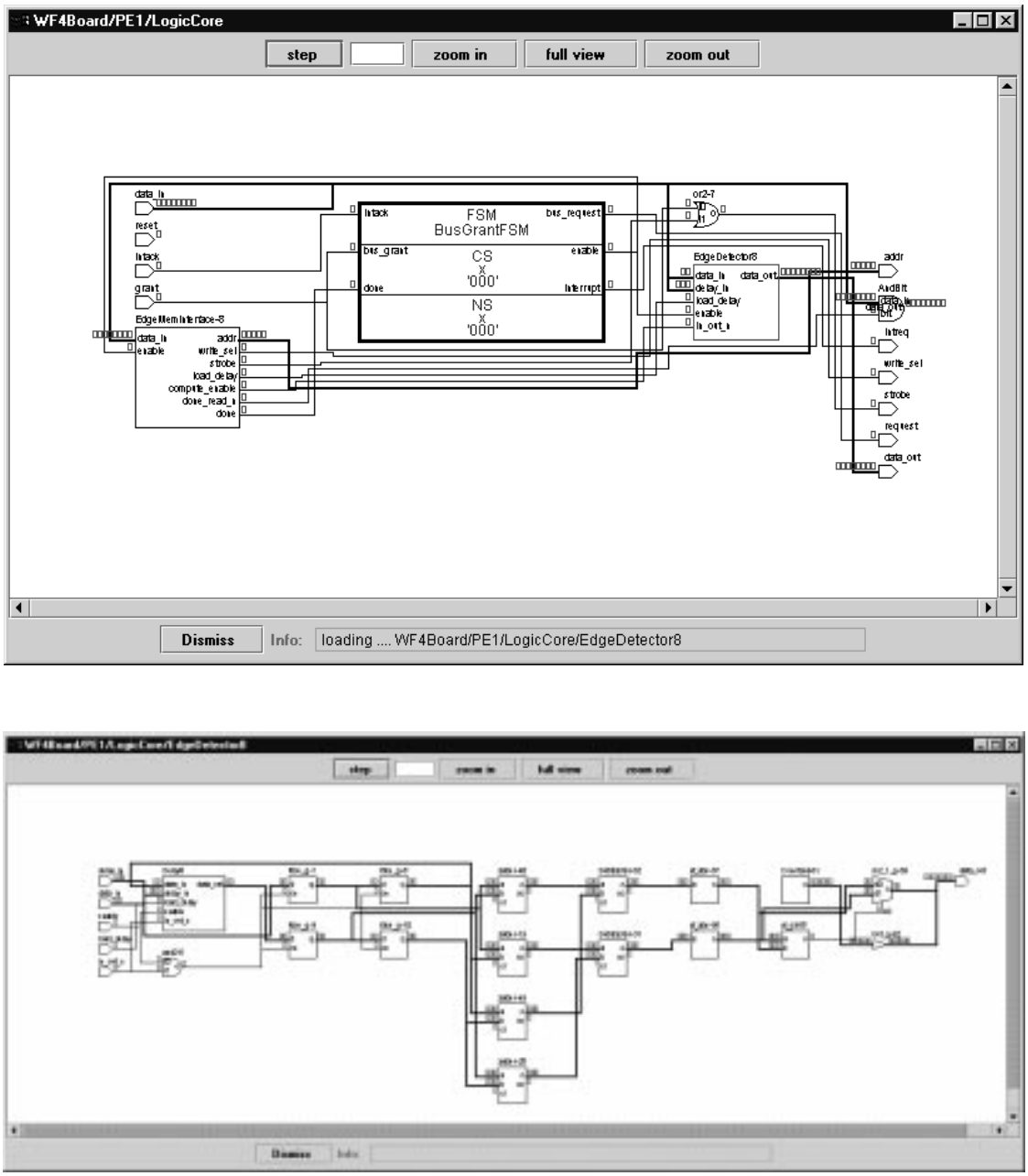

Figure 2-2: JAB Schematic Views.

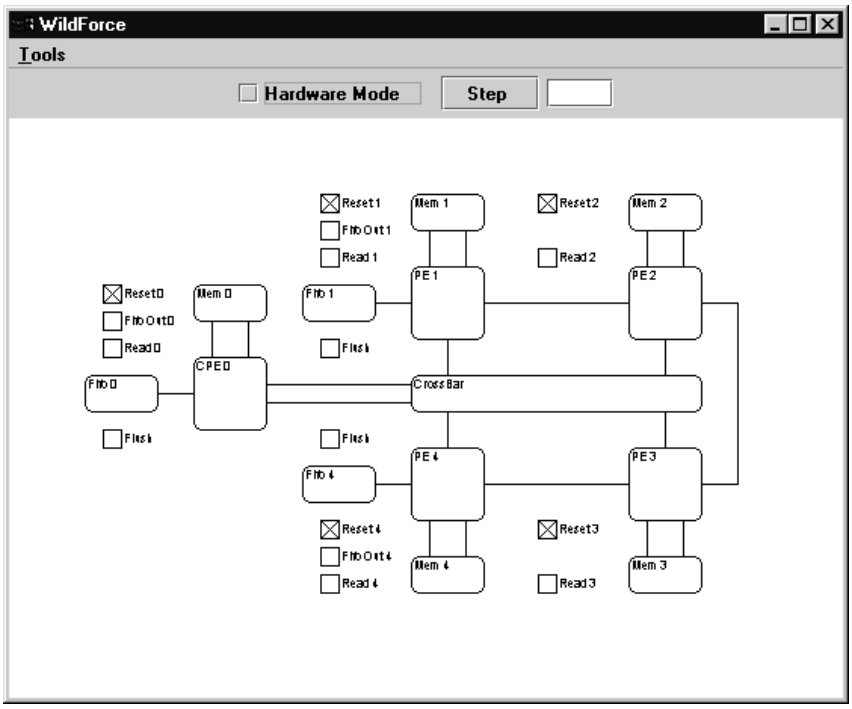

Figure 2-3: WILDFORCE Simulation Environment.

2.2.4.2 Structural Circuit Generation in JHDL

Development of structural logic in JHDL is performed by extending the `Logic` class. The `Logic` class is an abstract class that provides methods for creating hardware primitives. Listing 2-1 shows an example of a JHDL structural model of an absolute value operator.

The port interface between the operator and the rest of the structural circuit is described by the static string arrays that appear at the beginning of the class definition. In this case, the `portnames` class variable indicates that the `vt_abs` logic class has two ports, one called `a` and one called `o`. Some port interfaces have inputs and output with fixed widths. The `portwidths` class variable contains information about the bit widths of the ports. Elements with corresponding indexes in the `portnames` array refer to the same port. If the a port had a fixed bit width of 16, then the first element of the `portwidths` array would be the string "16". In this case, the bit widths of the ports are not known until the class is instantiated. This situation is indicated in the port interface by specifying the width of the port to be some string, called a generic from VHDL's similar mechanism. Ports that have generic widths are not bound to a specific width until the object's constructor runs. The `generics` class variable is used to specify an array of strings that contains a list of valid generic strings. Finally, the `portios` class variable specifies whether each port is an input or an output[4].

[4] The JHDL development team has revised the method for defining port interfaces since the original Janus work. A more intuitive interface serving the same purpose is now in place.

The port description is followed by an overloaded version of the `Logic` class' `getCellName()` method. The netlister requires all logical elements that have the same cell name have an identical netlist. This is to allow generation of reusable modules in the netlist and thus avoid creating a fully flattened netlist. Since this class is capable of creating absolute value operations with varying bit width inputs and outputs, the overloaded version of getCellName() simply appends the input bit width to a base name to guarantee that each width of absolute value operation has a unique name. The unique name is generated during the construction of the object.

The object's constructor assembles primitive logic to create complex behaviors. The constructor is typically divided into an initialization section and a hardware construction section. First, the constructor handles the generation of a unique name mentioned above. The next portion of the constructor checks to make sure that the widths of the wires passed into the object are of valid widths. The absolute value operator requires that the input and the output port have the same width. The constructor checks the widths of the wires passed to the constructor using the `Wire` object's `getWidth()` method. If the widths of the wires passed into the object are not valid, the object throws an exception to notify the user of the error. Next, the values of the object's generics must be bound to actual values. The `setGeneric()` method is called and is passed the width of one of the `Wire` objects that was passed into the constructor. The final stage of initializing the object is to link up the port names with actual `Wire` objects that represent them. This is done using the `port` method.

The hardware construction section follows the initialization section. Figure 2-4 shows a schematic representation for the circuit being built. An input bit vector, `a`, of some width *w* is the only input to the circuit. The `a` input is in two's complement representation. If the input is negative, the circuit must output the negated version of the input. The inverter and the adder produce this negated version of the input. The most significant bit of the input is used to determine the sign of the input and to control the multiplexor at the output. The multiplexor selects between the negated and non-negated version of the input.

JHDL takes a wire-centric view of circuit construction. In other words, the programmer creates and manipulates wires instead of creating and manipulating gates. The input and output wires that the circuit connects to are passed in through the constructor. The first thing that the hardware building section does is create a new wire called `negated`. This wire will correspond to the wire on the output of the inverter shown in the Figure 2-4. Wires can have multiple bits. The bit width of a wire is called when one is constructed. The logic class has a method called `wire()` that is an abstract factory[5] that produces `Wire` objects.

[5] An abstract factory is an object-oriented design pattern [GamH95]. A factory is a class or method that creates and initializes objects. An abstract factory returns a reference to an abstract base class that refers to a concrete object of an appropriate type. In this case, the `wire()` method creates a wire appropriate for the hardware being targeted. This allows the user to create a wire object that is independent of the target hardware.

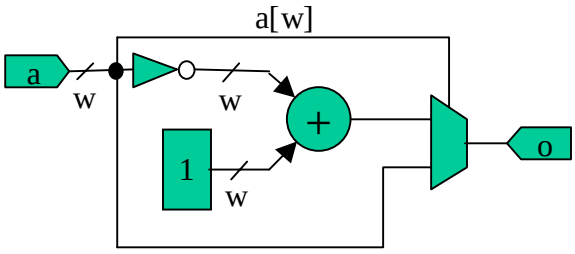

Figure 2-4: Schematic for Absolute Value Operator.

```
public class vt_abs extends Logic
{
      public static CellInterface[] cell_interface = {
            param(“w”,INTEGER);
            in   ("a", “w”),
            out  ("o", “w”)
          };

      private String celname;

      public String getCellName()
      {
            return celname;
      }

      public vt_abs(Node parent,Wire a,Wire o)
      {
            super(parent);

            bind(“w”,width);
            connect("a", a);
            connect("o", o);
            celname = cellname+"_"+width;

            if( a.getWidth() != o.getWidth() )
            {
                  throw new BuildException(cellname + ": input and output
width must match");
            }


            Wire negated = this.wire(this,width);
            add_o(not(a),constant(width,1),negated);

            mux_o(a,negated,a.getWire(width-1),o);
      }
}
```

Listing 2-1: Sample of JDHL Structural Logic.

Next, the constructor calls the Logic class' `add_o()` method. The Logic class has a large number of methods whose purpose is to create primitive logic structures. The methods connect input wires passed in from the method invocation, and connect them to an output wire that the user passed in or create a new output wire and return a reference to that wire to the caller. The user decides how to handle the outputs. Methods that have an `_o` appended to the end, like `add_o()`, connect the new structure's outputs to wires passed in by the user. Methods without the `_o`, like `add()`, create a new wire for their output and return a reference to it. Both methods can be seen in the line of JHDL that creates the negated version of the input. The `add_o()` method has three parameters. The first two are the addends and the third is a wire to which to connect the sum output.

From the schematic, it can be seen that the two inputs to the adder are an inverted version of the input and a constant one of an appropriate width. The expression that forms the first parameter to the `add_o()` call is a call to another Logic method named `not()`. The `not()` method creates a bus wide inverter structure. It is sized automatically by the width of the input wire passed to it. Note that the `not()` method does not have an `_o` appended to it, so it creates a wire for its output and returns it from the method invocation. This is convenient in this case since we do not really need to have a separate named wire for the inverted signal. This sort of syntax greatly simplifies circuit construction by allows complex expressions to be simply build up from simple operations.

The constant one value on the other input is generated using the Logic class' `constant()` method. Again, since the `constant()` method has no `_o` it creates a new output wire and passes it back. The `constant()` takes two parameters. The first indicates the bit width of the output and the second indicates what constant the user wishes to produce. Finally, the `negated` wire is passed into the `add_o()` method as the third parameter. The output of the adder is consequently connected to the `negated` wire.

The next line of JHDL creates the multiplexor that selects the output. One input is the original input `a` and the other is the newly created `negated` wire. The `mux_o()` method creates a bus wide multiplexor. The first two parameters are the signals to be multiplexed. The third is the select control and the fourth is the multiplexed output.

The select control parameter demonstrates the JHDL method for extracting single bits from bus wide wires. The `getWire(n)` method of the `Wire` class will return a 1-bit wire with the value `n`th wire from the object.

JHDL was chosen as the hardware description paradigm for this work because of its inherent flexibility for the application developer and the tool developer. It allows the application developer immense flexibility in terms of producing reusable adaptable hardware modules. It allows the tool developer to extend JHDL classes and utilize existing interfaces to provide tool integration that would be impossible using VHDL.

2.3 Dynamic Allocation of Hardware Resources

One of the goals for a complete RTR development system presented earlier was the ability to create, place and route new configurations at run time. Both of the search engines presented in Section 2.1 exhibit this behavior. The search engines implement this ability within a very limited context and in an application specific manner. The literature contains some research that investigates providing this capability in a general-purpose framework.

Burns proposes an operating system methodology for supporting dynamic creation of hardware configurations [BurD97]. He proposes exposing a low-level set of interfaces to the application programmer that encapsulates many of the tasks that would be required to support a dynamic run-time environment. Burns envisions an environment where configurable elements are shared between applications, may be partially reconfigured, and dynamic placement and routing of circuits at run-time is typical.

Burns begins with an argument for utilizing partial reconfiguration to leverage partial evaluation of computational elements. A detailed description of partial evaluation as it relates to partially reconfigurable hardware can be found in [SinH96]. Briefly, partial evaluation is a well-accepted technique in software compiler design where compilers automatically identify expressions that are constant and evaluate them at compile time to avoid run-time overhead. This same technique can be applied to hardware computations where one coefficient may be constant for a relatively long period. The taps in a typical FIR filter, for instance, have a relatively constant coefficient associated with them. Building a multiplication unit where one of the multiplicands is constant can have significant savings, in terms of configurable resources. In an ASIC design, if the coefficient can be changed at run-time a full multiplier must be built. The dynamic nature of a partially configurable FPGA, however, allows the multiplier to be replaced with an appropriate partially evaluated multiplier at run time. The ability to do this is one of the primary advantages of partial reconfiguration. The authors note the need for specialized place and route tools to support this kind of design and the need for run-time support, a means for which they later propose.

The need to allow multiple applications or threads to compete for resources on the configurable platform requires some sort of run-time system to handle contention. The authors propose a Virtual Hardware Manager (VHM) that fields requests from applications for configurable resources and interacts with other elements of the run-time system to try to service them.

Since applications are dynamically competing for resources, their circuit configurations cannot provide absolute placement of their components. Any area they chose for their absolute placement might already be in use by another application. Applications can merely provide a relative placement and allow the run-time system to perform the absolute placement. This is another service provided by the VHM. When the VHM receives a request to place a circuit on the configurable platform, it examines its operating tables to determine if there exists a region of unused logic where the circuit can be instantiated. If there is, the VHM can send the circuit to another operating system agent, called the Transform Manager, which performs the absolute

placement. The Transform Manager is also capable of rotating circuits if that is useful in placing them in an available region.

Dynamic placement and routing is an active area of research and involves finding acceptable NP-complete heuristics that can execute at run time and development of a sufficiently powerful programming interface for controlling FPGA configurations. Xilinx has recently released a Java API called JBits that provides the foundation for a useful configuration interface [Xili99]. Given the complexity that these problems add to a system, the Janus tools do not implement support for dynamic configuration creation. Given the low-level nature of the tools described by Burns, however, the structure of the Janus tools does not preclude the eventual inclusion of dynamic configurations when the technology has matured.

2.4 Janus Design Goals

Based results reported in the literature, previous experience with RTR design automation and a general pragmatic approach to tool development, the design of the Janus tools is governed by four goals.

2.4.1 Unified RTR Application Description

All of the applications examined in Section 2.1 were based on a CCM-accelerator architecture. The CCM-accelerator architecture is composed of two connected halves: the host side and the CCM side. The host side is generally a microprocessor-based workstation. The CCM side is a computational array of FPGAs. Some sort of local bus typically connects the two, although this connection could be any kind of communications network. All of the applications in Section 2.1 were designed in two parts. The host side was written using a standard programming language with calls to specialized libraries for interfacing with the CCM. The CCM side was created either using some sort of ASIC-style design technique, schematic capture or synthesis of VHDL. The two design efforts are completely separate. If an application uses reconfiguration to leverage virtual hardware, the design fragments even further. None of the existing design tools provide an abstraction for hardware designs with multiple configurations. Designers are forced to create designs that, as far as the development tools are concerned, are totally unrelated.

Janus' primary design goal is to allow RTR application developers to specify their designs using a single design flow. One of the primary reasons for selecting JHDL as the structural representation for the CCM side is to place the circuit design in the same design space as the host side. VHDL is not a convenient medium to write user interfaces or to provide database access, but Java is. By bridging the gap between the host side and the CCM side, the Janus tools not only provide a unified design description for an application they provide a means for the two designs to interact.

2.4.2 Time-Multiplexed Virtual Hardware

The automated design tools examined in Section 2.2 all have one fundamental limitation. The size of the problem that they can address is limited by the amount of configurable resources available on the CCM side. Run-time reconfigurable applications do not have this fundamental limitation as demonstrated by the ATR and image interpolator applications described in Section

2.1.2. The ATR and image interpolation applications demonstrate that temporally multiplexing the FPGA resources can significantly increase the scope of designs. The Janus tools provide an abstraction of this process that allows users to easily specify reconfigurations within the unified design.

2.4.3 A Pragmatic Approach

Many lessons were learned from the aborted tool design described in [HudL98]. One of the most important was that the Janus tools are exploring a new area of design space. So much is unknown about the design of RTR applications that it is easy to become engaged in an effort to solve problems that, after much effort, turn out to be unrelated to the core design issues. Therefore, the Janus tools take an extremely pragmatic approach to development. The initial version of the tools, documented within this dissertation, provides a fully functional working design system. This fully functional system was made possible by a steadfast belief that a working tool would provide a stepping stone to understanding which further problems should be explored. Along the way, opportunities to solve many different NP-complete problems were put aside until a firmer understanding could be developed of which problems were worth solving.

2.4.4 Architecture Independence

Given the expressed goal of providing a pragmatic set of tools that future researchers can leverage, the Janus tools would be useless if they were bound to a particular architecture. Not only is there a wide variety of CCM platforms in use by the research community, the multi-configuration architectures that may best support RTR applications are not even available at this time. Further, the advantages of being able to port applications between different CCMs are reason enough to create an architecture independent system. Therefore, it is one of the goals of the Janus tools to be able to design applications that can be automatically targeted to different CCMs.

Caution must be taken, however, to insure that in making applications portable, their ability to exploit the advantages of the underlying architecture is not hindered. This requirement drives some of most interesting features of the Janus design tools and provides some of the most fascinating avenues for further exploration. Applications built with the Janus tools must automatically adapt to exploit their target architecture.

Taken together these goals motivate the design of the Janus tools. Chapter 3 discusses the design in detail.

Chapter 3 Janus

The primary contribution of this dissertation is an object-oriented framework that allows RTR applications to be developed using an integrated design representation of host and CCM and without knowledge of the target architecture. One of the central ideas of object-oriented design philosophy is that an object's interface should be separate from its implementation. This idea provides the basis for creating architecturally independent designs. The Janus tools provide an abstract environment within which to design RTR applications. The interfaces within the abstract system can be mapped onto actual behaviors in existing CCMs. The specifics of that mapping are part of the implementation and therefore hidden from the user of the tools. There is a wide range of ubiquitous software development tools that could be used to implement the Janus abstraction. Any object-oriented language that is capable of linking to CCM vendor API libraries is a suitable environment. Such tools include popular programming languages like C++ and Java. This section discusses the Janus abstraction[6] and the motivations behind it.

The most basic motivation for the creation of any development environment is to satisfy the needs of some set of users. Figure 3-1 is a use case diagram for the Janus tools. It depicts four roles that users might assume. The first, in the end perhaps the most important, is the end user. When all is said and done, there must be some intended user for any application. In this case, the user is interested in utilizing an FPGA accelerated computational application. They are not interested in compiling it, designing it or building it. Therefore, the tools must be capable of delivering some sort of compiled version of an application to the user that behaves very similarly to a standard all-software solution.

The most obvious user of an application development system is, of course, the application developer. The skill set of an application developer may vary from being a domain expert who is building an application as a tool to assist them in other efforts, to a professional developer who is extremely knowledgeable in the use of the tools themselves. Neither extreme is interested in

[6] Discussions of object-oriented design have lacked, until recently, a well-accepted coherent notation. The object-oriented design community has recently standardized such a notation called the Unified Modeling Language (UML). The UML is a graphical language with well-defined semantics. Complete discussions of UML notation can be found in [BooJ98], [RumJ98], [JacB98], and [FowS97]. All of the diagrams in this dissertation that describe object-oriented constructs are presented in UML.

having to develop a detailed understanding of a particular RTR CCM architecture. Neither one is interested in learning the specific application programming interface (API) associated with a target CCM. Neither is particularly interested in the internal workings of the compiler, except as it directly constrains their design options and affects the performance of their application. These requirements provide yet another motivation for architecture independence in a set of design tools. If the application developer should not have to understand the underlying API for the CCM, the design model the developer is working with cannot be based on a particular architecture.

Even if the application developer does not have to understand the detailed workings of the target hardware, the tools themselves have to capture this knowledge in some way. This creates the need for the third actor in Figure 3-1, the hardware backend developer. This actor is part of the tool development effort. When new architectures are developed and new FPGA technologies are produced, someone has to create a new backend for the development environment that allows it to use the new hardware. The backend developer fulfils that role. The Janus abstraction uses the inheritance mechanism of object-oriented system design to minimize the efforts of the backend programmer.

One last actor is portrayed in the use case in Figure 3-1. Since the Janus tools are an evolving system, with an eye toward future expansion, care has been taken to design the tools in a modular way that allows future researchers to experiment with replacing and improving various portions of the compiler and run-time environment. The various portions of the system have dependencies, but some portions of the system can be changed without breaking others. New schedulers, for instance, can be written without affecting the run-time system or the application model. Changes to the run-time model will sometimes require changes to the scheduler. Adding features and functionality of any significant kind to the application model will almost certainly require changes to the run-time system and the scheduler.

A more detailed view of the structure of the abstraction itself is presented in Figure 3-2. Figure 3-2 is an UML package diagram describing the division of the classes in the Janus system among the various packages and the dependencies between the various Janus packages. Also diagramed are the dependencies between user applications, shown in the **apps** package, and the Janus packages.

The **hardware** package contains classes that provide abstractions of hardware architectures, processing elements and memory. The **runtime** package provides data structures for describing the sequence of run-time events and interfaces to the hardware architecture for executing them. The **schedulers** package contains the compile-time code for translating the user's description of their application into a sequence of run-time events. Note that user applications are not dependent upon the **hardware**, **runtime**, or **schedulers** packages. Applications are independent of the underlying hardware model, compiler components, and run-time structures.

The **stages** package defines the interfaces to the components that the application developer will use to model their application. The **base** package contains the application framework that the

user extends to create their own application. It defines the interface between Janus' compiler and run-time system and the user's application. Finally, the **app** package contains the code for creating Janus' user interface.

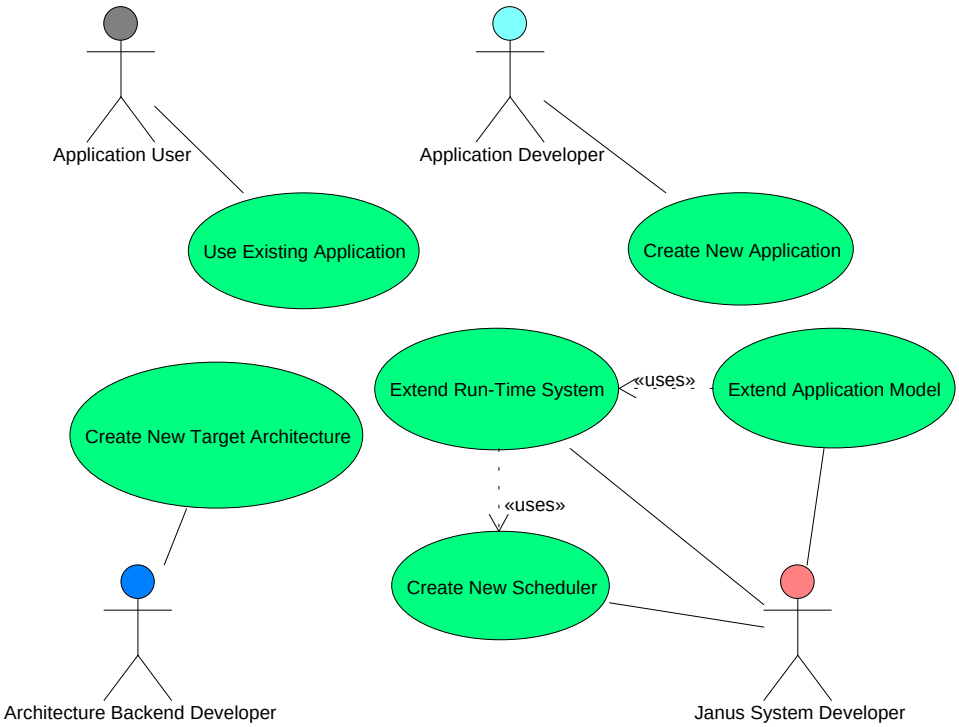

Figure 3-1: Use Case Diagram for the Janus Toolset.

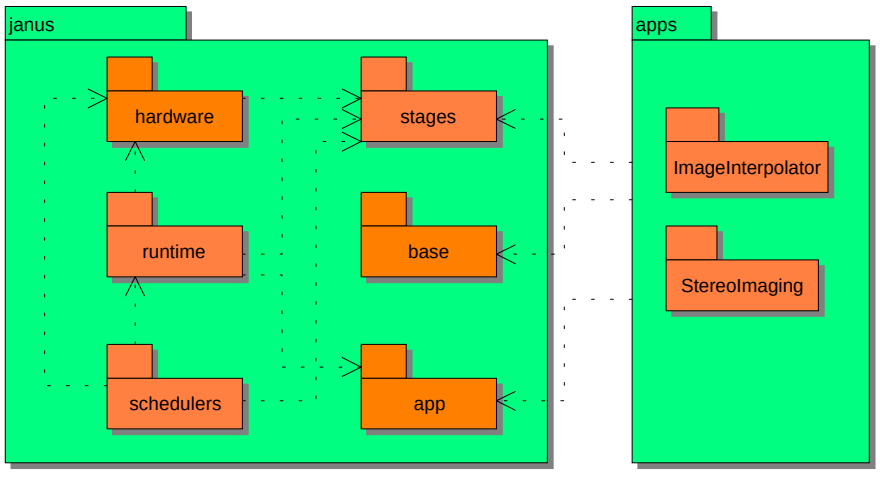

Figure 3-2: Package Diagram for Janus Abstraction.

This discussion proceeds by examining how the Janus abstraction addresses the use cases presented in Figure 3-1. This approach is meant to focus attention on the motivation for the abstraction rather than the implementation of it. The implementation will be addressed in Chapter 4.

3.1 Application Development

The development use case drives the design of the abstraction. Figure 3-3 shows an extended version of the application development use case. The overall task can be broken down into several relatively well-defined subtasks. The developer must create an application model that defines the sequencing of the computation and the nature of the computation itself. If the application has elements that are implemented in software or the developer wants to prototype certain portions of the design in software, software stages are created. Portions of the design that require hardware-accelerated operations must be designed and implemented. Some type of user interface must be built. The Janus toolset does not place any requirements on the nature of the interface. User interaction could be provided as a graphical user interface (GUI), a command line interface (CLI), or even as a network service. Finally, the finished application must be compiled so that it can be executed on a target platform.

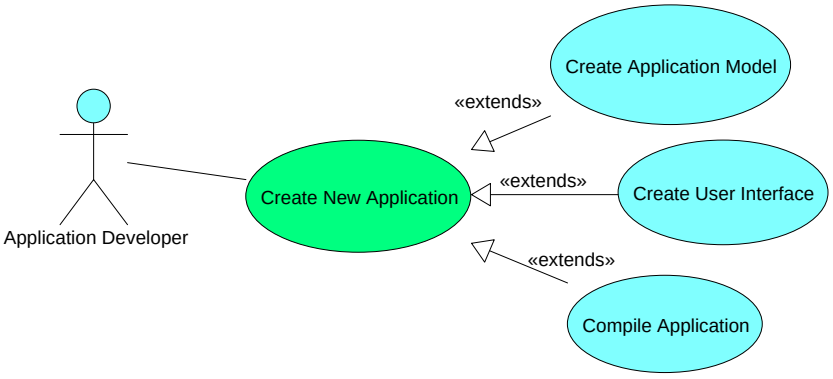

Figure 3-3: Extended Application Development Use Case.

3.1.1 Creating the Application Model

The **stage** package defines the basic structures the user will use while carrying out the "create application model" use case. The design of the **stage** package is highly influenced by previous work on a more ambitious toolset described in [HudL98]. Work on this original set of tools was suspended because the challenges involved were too difficult for a first attempt at a working set of tools. Some important lessons were learned concerning the relationship between real-world applications and the limitations of the run-time reconfigurable environment.

3.1.1.1 Initial Approach

The original toolset was based on a set of assumptions that experimentation revealed were poorly formed. The original tools were designed around applications that formed deep computational pipelines. The idea was that these pipelines were too large to be instantiated on the hardware

platforms typically available and that some form of temporal partitioning was required to break the problems into separate configurations that could be sequentially implemented on a run-time reconfigurable hardware platform. In practice, it was discovered that applications like this are rare. Typical applications that appeared to be of interest did indeed require more hardware resources than are typically available on RTR CCMs, but they did not form the extremely deep pipelines that the tools expected.

It would have been possible to represent the applications as extremely deep pipelines, but in such a representation, there was usually a section of the dataflow description where a huge amount of hardware would have been required to reorder the data. The image interpolator application is an example. The first part of the computation performs the inverse filter on the rows of an input image. The second part of the computation performs the same filter on the columns of the image produced by the first part of the computation. This data dependency means that the second part of the computation cannot begin until the first part is complete. In a dataflow representation, this means that the entire output of the row computation must be routed through a complex delay structure that reorders the data into columns. In an FPGA hardware implementation, this would require delays, built either from registers or from RAM primitives that could buffer the entire image. Furthermore, since the purpose of the delay portion of the dataflow is to reorder the data, it makes no sense to temporally partition anything within the delay structure. If it were temporally partitioned, the intermediate results would have to be stored in a RAM between during reconfiguration. Once in RAM, the next stage could simply read the data out of RAM in the order that it wanted. In practice, it is simply much easier to utilize the random access nature of an attached SRAM to store and reorder the intermediate data.

3.1.1.2 Janus Staged Approach

Based on this observation and others made from looking at applications like wavelet compression [CohD92] and SAR imaging [CasP98], the approach taken in the Janus tools is based on *stages of computation*. This approach allows applications to be described in the natural divisions that occur between significant reorderings of data. The image interpolator, for example, breaks down into four stages: inverse filter on the rows, inverse filter on the columns, Fast-Spline Transform on the rows, and Fast-Spline Transform on the columns. Intermediate results are retired to memory between each stage. This allows the following stage to simply read its data from memory in the order that it requires. No explicit data reordering is necessary.

Given this computational model, the Janus abstraction defines a set of container objects called stages. The stage containers allow the application developer to group together the computational operations that form an application. A class diagram of the **stages** package is given in Figure 3-4.

The package defines an interface, called ***Stage***, and three implementations of that interface, **StageSoftware**, **StageOrdered**, and **StageUnordered**. These three classes, and another interface called **Operation**, form the basic building blocks that an application developer uses to model their computation. The developer describes their computation by creating a hierarchy of computational stages. The **StageOrdered** object maintains an ordered list of references to other ***Stage*** objects. Each **StageOrdered** object has at least one reference to another ***Stage***. This

allows the user to create a tree of ***Stage*** objects. The leaves of this tree structure are instances of the **StageSoftware** and **StageUnordered** classes. The **StageSoftware** class allows the user to create software implementations of computational stages. **StageUnordered** is a collection class that allows the user to build up a set of hardware-accelerated operations that need to be performed during a given stage. These hardware-accelerated operations are implementations of the **Operation** interface.

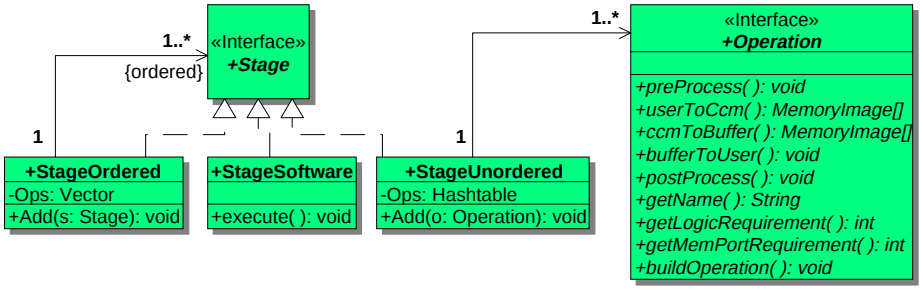

Figure 3-4: Class Diagram of the Stages Package.

Figure 3-5 is an object diagram showing an application model of the image interpolator discussed in Chapter 1. The structure is an ordered hierarchy. Each **StageOrdered** object maintains an ordered list of its aggregate ***Stages***. Each ***Stage*** associated with a **StageOrdered** object is another **StageOrdered** object, which adds another layer of hierarchy, or one of the leaf ***Stages***, **StageUnordered** or **StageSoftware**. Since all of the hierarchical objects maintain an ordering of their aggregates, there is an overall ordering maintained on the leaf nodes. If we assume that the association markers in Figure 3-5 are ordered spatially from left to right in the diagram, the overall order of the leaf nodes is **IF Rows**, **IF Columns**, **FST Columns** and **FST Rows**. The overall ordering defines the sequence of events that will happen at run time.

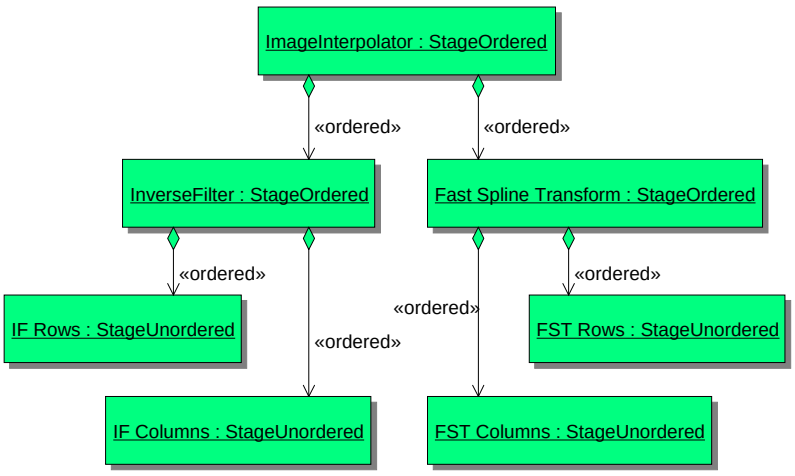

Figure 3-5: Object Diagram of Image Interpolator.

The ability to specify the overall ordering in a hierarchical way is a convenience provided for the application developer. The Janus compiler deals only with the overall ordering not the hierarchy. Future versions of the compiler may be able to take advantage of the information provided by the hierarchical representation to aid in making some design decisions[7]. A detailed example of the process of building a computational hierarchy can be found in Section 4.1.

Each of the leaf nodes in an application model describes a set of events that will occur at run time. In the case of **StageSoftware** objects, this run-time behavior is specified as a concrete implementation of **StageSoftware**'s *execute()* method. This ability to implement software stages in software provides the ability to create a rapid prototype of a system before implementing hardware-accelerated objects. As hardware versions of each stage are created, they can be easily substituted for their software implementations. It also provides the ability to incorporate software implementations of algorithms that are not well suited to CCM development.

StageUnordered objects contain an unordered set of hardware-accelerated operations. Operations are entities that span the boundary between the host computer and the CCM. These operations are implementations of the **Operation** interface shown in Figure 3-4. They have both a software component and a hardware component. Operations can be thought of as hardware accelerated processes. All operations are associated with at least one memory interface. Operations read their input data through a memory interface and retire their results back through the same interface or possibly another memory interface. Operations that are grouped together in a **StageUnordered** collection should all be computationally independent of each other. If two operations are computationally dependent upon one another, they should be relegated to separate **StageUnordered** objects that are sequenced in the appropriate order by a higher-level **StageOrdered** object.

On the software side, the operation implementation describes how data is transferred between the host-side application and the processing elements on the RTR CCM. It also has some methods that allow the compiler to query information from the operation concerning its name and resource requirements. The *preProcess(), userToCCM(), ccmToBuffer(), bufferToUser(),* and *postProcess()* methods are all used to describe how data should be moved from the user's application to the target hardware. The *preProcess()* and *postProcess()* methods are used to take care of any host-side processing necessary to set up data before or clean up data after an operation executes. The *userToCCM()* method sets up the transfer of input data from the host-side of the application to a processing element on the CCM. The *ccmToBuffer()* and *bufferToUser()* methods are used to set up the retrieval of data from the CCM returning it to the software portion of the application. Two methods are required to handle the retrieval of the data

[7] In some ways, the current application model is unnecessarily constrained. The image interpolator application shown in Figure 3-5, for instance, does not require that **IF Rows** precede **IF Columns**. Nor does it require that **FST Columns** precede **FST Rows**. Either pair of stages may be executed in either order. This implies that the design hierarchy could include a modified version of the **StageUnordered** class as hierarchical nodes. Inclusion of unordered hierarchical nodes would allow the user to express computational independence like that found in the image interpolator. This would leave the decision of which stage to do first to the compiler.

because typical CCMs do not provide high-speed random access to their processing element's memories. This issue is discussed in more detail in Section 4.4.

The *getLogicRequirement()* and *getMemPortRequirement()* methods provide information about the hardware resource requirements of the operation. These methods are used by the scheduler to assign operations to processing elements. This process is discussed in detail in Section 5.3.

The *buildOperation()* method provides the hardware description of an operation. The *buildOperation()* method uses JHDL to build a structural model of the operation that can be netlisted and placed-and-routed on the target FPGA. More details on using JHDL to build operations can be found in Section 4.5.

3.1.2 Creating the User Interface

Creation of the user interface is a highly application dependent and implementation dependent procedure. The Java implementation of the Janus abstraction exposes certain interfaces for communication between a user interface and a Janus application, but no explicit support is provided for building the interface itself. The Java language itself is very useful for creating graphical, command line and even network interfaces. Java was used to create the user interfaces for the applications presented later in this work. Screenshots of some of the interfaces can be found in Chapter 6.

3.1.3 Compiling the Application

A basic understanding of the Janus hardware model is necessary to understand the compilation process. Figure 3-6 presents a class diagram that depicts the hardware abstraction. The abstract class ***HardwareArchitecture*** provides a software interface to the target RTR CCM. A ***HardwareArchitecture*** is composed of one or more associated ***ProcessingElements***. While not explicitly represented with a separate class, ***ProcessingElements*** may each have one or more memory ports associated with them.

A concrete implementation of a ***HardwareArchitecture*** models the behavior of a specific hardware platform. For instance, the **WildForceXL_4062** class models the behavior of an Annapolis Micro Systems WILDFORCE XL populated with XC4062s. The **WF4_4062** class models the processing elements available in the **WildForceXL_4062** class. Figure 3-7 shows the relationship.

HardwareArchitectures and ***ProcessingElements*** both have methods that allow the compiler to query information about the available hardware resources and to assign hardware operations to memory ports on individual processing elements. These methods allow the compiler to interact with the architecture and the processing elements to determine which operation should be scheduled on which element. The compiler acts within the constraints of available configurable logic and memory ports. Compilation involves the most complex interactions between the classes in the Janus abstraction. Figure 3-8 is a class diagram showing the classes and associations involved in the process.

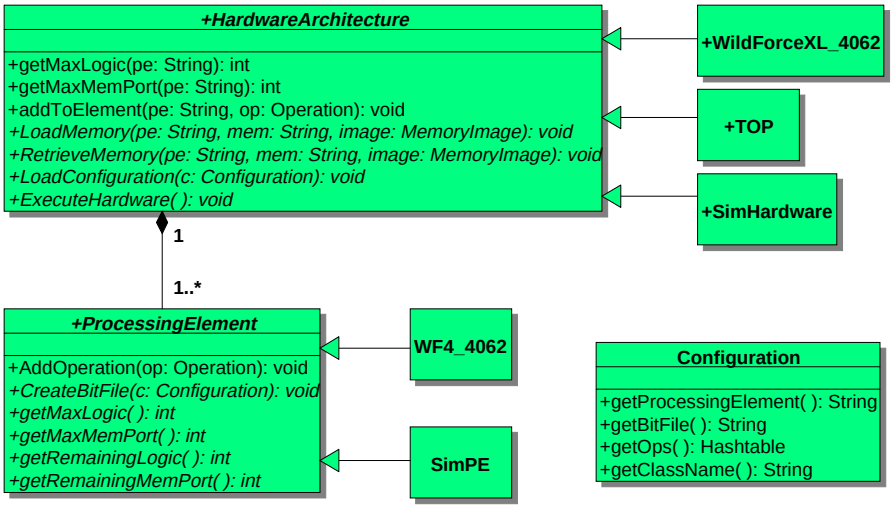

Figure 3-6: Janus Hardware Abstraction.

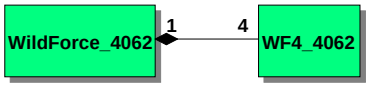

Figure 3-7: Concrete WILDFORCE Model.

In the current manifestation of the Janus tools, the ***Scheduler*** is the only compilation class. Future versions of the tools may provide more advanced compilation facilities. The basic contract of the ***Scheduler*** is to take a reference to an application model and a ***HardwareArchitecture*** and produce a **RunTimeScript**. A **RunTimeScript** is an ordered list of events that need to occur at run time during the execution of the application. The run-time events are represented by the various ***RunTimeEvent*** classes. The run-time events currently supported by the Janus abstraction are shown in Figure 3-8 as the concrete inheritors of the ***RunTimeEvent*** class.

The **LoadConfiguration** event causes a set of hardware configurations to be loaded onto the processing elements of the hardware architecture. In the specific case of the WILDFORCE board, this corresponds to programming one or more of the Xilinx FPGAs with appropriate bitfiles.

The descendents of **MemoryEvent** both represent operations on the memories associated with the processing elements on the CCM. A **FetchMemory** event causes data to be copied from somewhere in the host application to a particular location in a particular memory on the CCM. A **RetireMemory** event causes data to be copied from one of the memories on the CCM to somewhere in the host application. These events are used to provide input data to hardware operations and to retrieve results once the computation has been accomplished.

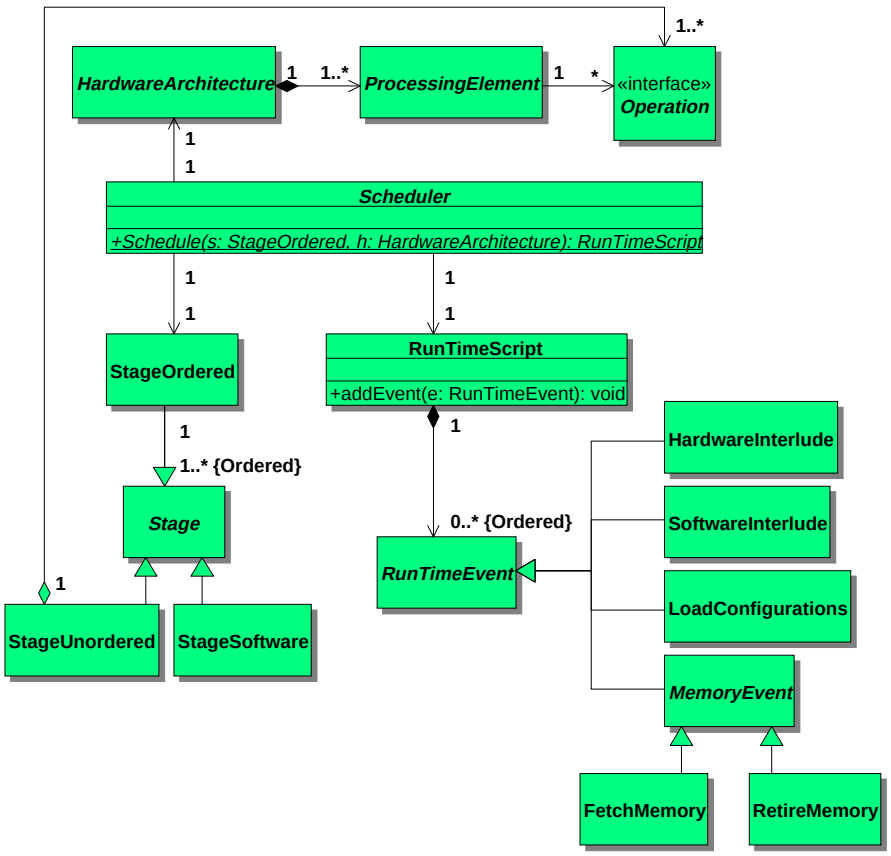

Figure 3-8: Interaction of Classes during Compilation.

The **HardwareInterlude** and **SoftwareInterlude** classes both represent the run-time execution of the computation represented by a ***Stage***. A **HardwareInterlude** corresponds to the execution on the CCM of some subset of operations from a **StageUnordered**. This involves starting the clock on the hardware platform, waiting for a signal from the CCM indicating that the computation has been completed, and stopping the clock. Execution of a **SoftwareInterlude** corresponds to calling the *execute()* method of a corresponding **StageSoftware**.

The first step performed by the scheduler, when given an application model and hardware architecture, is to flatten the hierarchy in the application model and produce a list of the computational stages that need to be executed. Next, it creates a **RunTimeScript** with no associated **RunTimeEvents**. Then, the scheduler processes each stage in the flattened list, one by one.

If the next stage is a **StageSoftware** object, the scheduler creates a corresponding **SoftwareInterlude** object, adds it to the **RunTimeScript**, and continues with the next ***Stage***. If the next stage is a **StageUnordered** object, the scheduler's job is more complex.

Consider the simplest case. The **StageUnordered** object has some number of associated **Operation** objects. The simplest case for the scheduler is when all of the hardware operations in a stage utilize the same hardware implementation. The **IF Rows** unordered stage from the image interpolator is an example of this case. All of the operations in that stage are inverse row filter operations. An instance of an **IF Rows** stage might have 128 **Operation** objects, each of which represents performing an inverse filter on a row of an input image. The hardware required to perform each of these operations is identical. A time-area tradeoff can be made in evaluating the operations. All 128 **Operation** objects could be executed using a single hardware instance of the operation. The same hardware would just be reused repeatedly and the overall execution time would be 128 times the length of time required to evaluate a single inverse filter. That option trades off time for reduced hardware requirements. Alternatively, if enough configurable hardware is available, 128 instances of the inverse filter could be implemented in parallel and the execution time of the 128 inverse filters could be reduced to the execution time of a single filter. The goal of the scheduler is to create a run-time series of events that minimizes the execution time of the application. Since there is no cost associated with using the available hardware on an RTR CCM, it makes sense to trade as much area for time as possible.

If a **StageUnordered** stage is constrained to a single operation type, the scheduler must simply figure out how many instantiations of the single hardware operation it can fit onto the RTR CCM. The scheduling problem becomes a simple packing problem where a set of identical objects must be packed into a set of identical containers. This problem can be solved optimally in linear time. The algorithm for doing this is described in Section 5.3. Figure 3-9 is an example of the results of such a packing. This operation allows the scheduler to adapt the application to take advantage of the resources available on a target architecture.

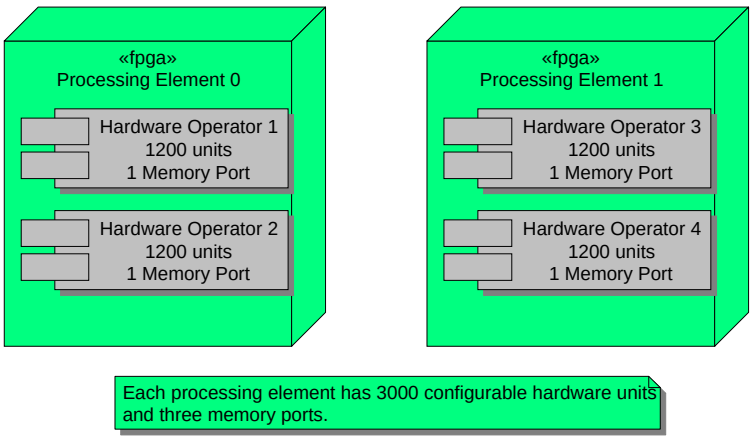

Figure 3-9: Sample Hardware Configuration.

Figure 3-9 illustrates a target platform with two processing elements, each of which has three memory ports and 3000 units of configurable hardware. The results shown are a packing of inverse filters onto the platform. The filter requires 1200 units of configurable hardware and a

single memory port. Given the packing shown, both FPGAs have one free memory port and 600 unused hardware units. No further instances of the hardware operation can be allocated.

Having created hardware instantiations of the filter, the scheduler interacts with the hardware architecture to produce a set of netlists for further processing by FPGA vendor tools. This processing produces a set of configuration files that can be downloaded to the hardware platform. At run time, this configuration will be loaded onto the target hardware and used to evaluate the operations.

Now that the scheduler has created a hardware configuration, it can schedule the 128 **Operation** objects from the **IF Rows** stage to execute on the hardware instantiations. The scheduler builds a run-time series of events that follows the basic computational cycle depicted in Figure 3-10. A hardware configuration is loaded on to the CCM, input data is provided for the next set of operators scheduled to execute, the operations are carried out, the results are retrieved, and, if necessary, the cycle returns back to the memory initialization step to execute the next set of operators. Since the operations all use identical hardware, no reconfiguration is necessary.

For the example architecture shown in Figure 3-9, the scheduler determined that four hardware operation units could be run in parallel. Scheduling four of the **Operation** objects to the hardware instantiations during each computational cycle, the entire stage can be completed in 32 times the length of time it takes to compute a single filter operation. If the target architecture allowed more instances of the hardware operations to be configured, it would take proportionately less time to compute the stage.

The **BasicScheduler**, described in Section 5.3, implements a scheduling algorithm using the single operation type constraint. During development of the Janus implementation it was simply much easier to only deal with algorithms that were easily determined to not be NP-complete. Future versions of the tools could incorporate more complex scheduling algorithms. The basic scheduling process is diagrammed in Figure 3-11.

3.2 Application Execution

To be of any value, after an application has been successfully compiled, it needs to be executed. The Janus abstraction specifies a run-time environment that executes the series of ***RunTimeEvents*** generated by the compiler. The Janus implementation provides support for graphical user interfaces. A discussion of that support is beyond the scope of this dissertation. In this section, only the object model relating to run-time execution on the CCM will be discussed.

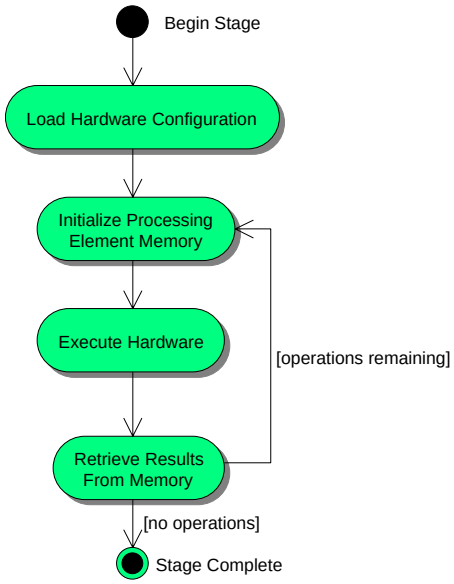

Figure 3-10: Activity Diagram for Compute Cycle.

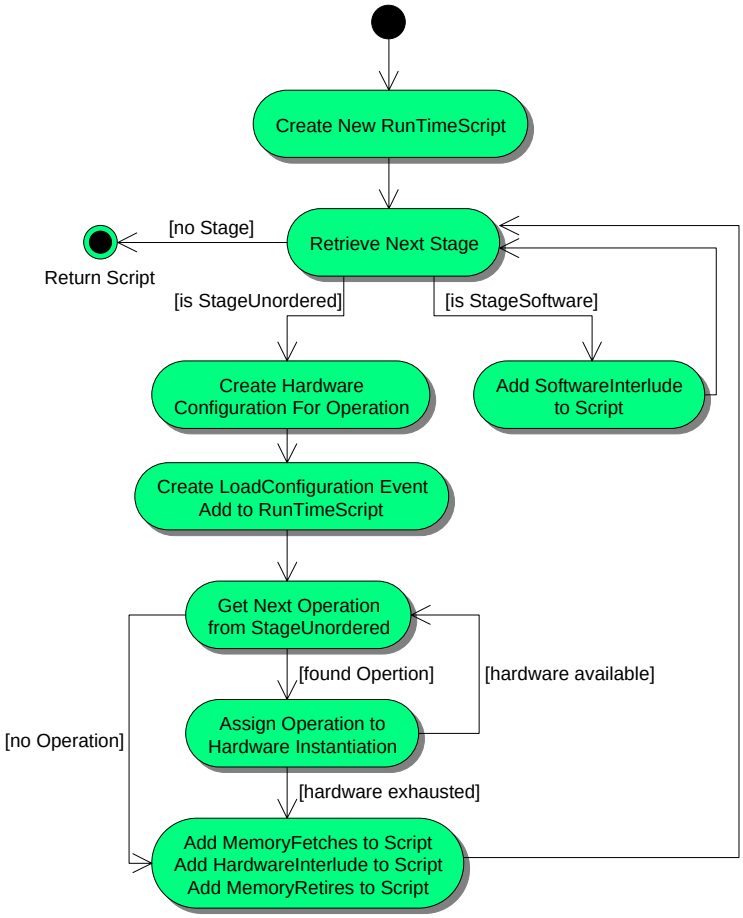

Figure 3-11: Basic Scheduling Algorithm.

The compiler has produced a set of netlists that describe the required hardware configurations. The user must process the netlists and make them available to the Janus run-time system. The run-time system also requires the series of run-time events that the compiler produced. The compiler serializes the sequence to a file. The combination of the run-time sequence and the hardware configurations constitute the "object code" of a Janus application.

Part of the run-time model has already been discussed in Section 3.1.3. Figure 3-10 gives a high-level view of the run-time execution model. Figure 3-12 is a sequence diagram that describes the process in detail.

The **:RunTime** object is created and controlled by the Janus user interface. Figure 3-12 show the initial sequence of events that occur when an application is executed. The **:RunTime** object iterates over the sequence of event objects contained in the **:RunTimeScript** object. The events are translated into method calls to the **:HardwareArchitecture** object. Several applications have been implemented using the Janus compiler. Descriptions of the applications can be found in Chapter 4 and Chapter 6.

Figure 3-12: Application Execution Sequence Diagram.

Chapter 4 Anatomy of an Application: Image Interpolation

This section presents an application model of the image interpolation application within the reference implementation for the Janus design abstraction. In keeping with the spirit of architecture independence, the standard programming language chosen for the reference implementation is Java [GosJ98]. Java provides a portable platform for hardware development. While Java itself executes in an interpreted mode on virtual hardware, the language standard provides a means for executing special purpose code on the native system. The interface is called the Java Native Interface (JNI) [GorR98]. This interface allows a small amount of native code to provide an interface between the host system and the CCM's API.

4.1 Image Interpolation

The theoretical basis for the image interpolator is due to Ferrari [FerP97]. The interpolation procedure increases the resolution of an image by deriving new pixels between the existing ones. Typically, this kind of interpolation is done with cubic splines, but the computational requirements of that method are prohibitive for real-time applications and generally require floating-point arithmetic to maintain acceptable resolution. Bilinear interpolation is a computationally tractable solution, but the resulting image quality is unacceptable for many applications. The 2-5-2 spline approach presented in [FerP97] provides better results than the bilinear method with roughly the same computational complexity as the bilinear approach.

The 2-5-2 spline matches the lower computational complexity of the bilinear approach by choosing its spline basis so that the coefficients in the computation of the spline are powers of two. Since multiplication and division by powers of two can be implemented with low-resource extremely fast shift units, computation of 2-5-2 splines is dramatically faster and less resource intensive than other methods.

The interpolation procedure is broken down into two parts: the inverse filter (IF) and the Fast Spline Transform (FST). Both computations must be performed in two dimensions, i.e. on both the rows and the columns of the image. Calculations in both directions require pixels from only one row or column at a time. The computation can be performed on the rows first and then the

results from that computation can be used in the column calculation. The inverse filter takes the original image and calculates the spline vertices according to the following:

$$V_n = \sum_{k=n-l}^{n+l} (-\frac{1}{2})^{|k-n|} P_k$$

where n varies from one to the number of pixels in a row or column and P_k's for $k<1$ *and* $k>m$ are zero. This process is diagrammed in Figure 4-1.

The output from the vertex calculation is an image that is approximately the same size as the input image. The vertex calculation has some edge effects that make the image slightly larger than the original. This image is expanded to a higher resolution by introducing some number of new pixels between the spline vertices. The FST phase takes the spline vertices and uses 2-5-2 spline arithmetic to interpolate values for the new pixels. This procedure is diagrammed in Figure 4-2. The result is an image with higher resolution than the original.

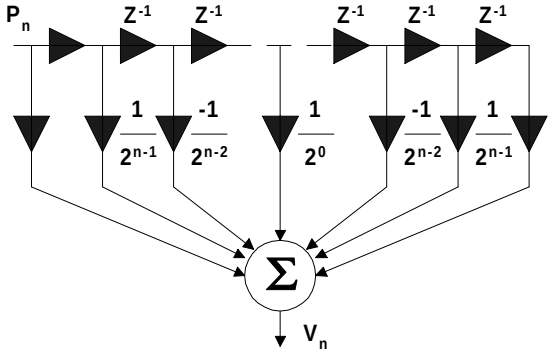

Figure 4-1: The Inverse Filer [FerP97].

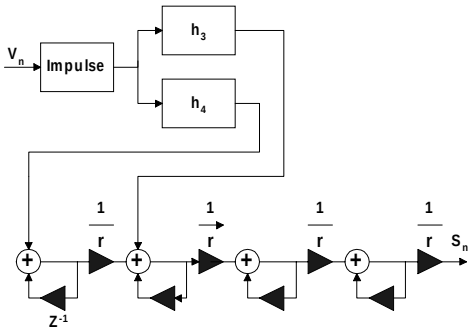

Figure 4-2: The Fast Spline Transform [FerP97].

4.2 Approach

The implementation of the image interpolation process is an excellent example of using RTR methods to map a large calculation onto the relatively small Xilinx Reference Board. The mapping process began with an analysis of the interpolator's computational structure.

The authors of [FerP97] demonstrated their interpolation engine using MatLab. Based on their MatLab code, a C++ based engine was created that implemented the interpolator using integer arithmetic. The C++ engine was used to verify the procedure using integer math and to determine the bit precision required to achieve acceptable quality in the output image.

4.2.1 Computational Structure of Image Interpolation

Figure 4-3 illustrates the computational structure of the image interpolation process. Each gray circle represents either a row or column in the working image. The rows and columns of the inverse filter are separable and thus mutually dependent on each other. In other words, all the columns must be computed before any of the rows can be computed or vice versa. Whether the rows or the columns are computed first is not significant. The rows and columns of the Fast Spline Transform are also mutually dependent. In addition, the diagram indicates the FST is completely dependent on the results of the IF. All the rows and columns of the IF must be computed before any of the rows or columns of the FST can be computed.

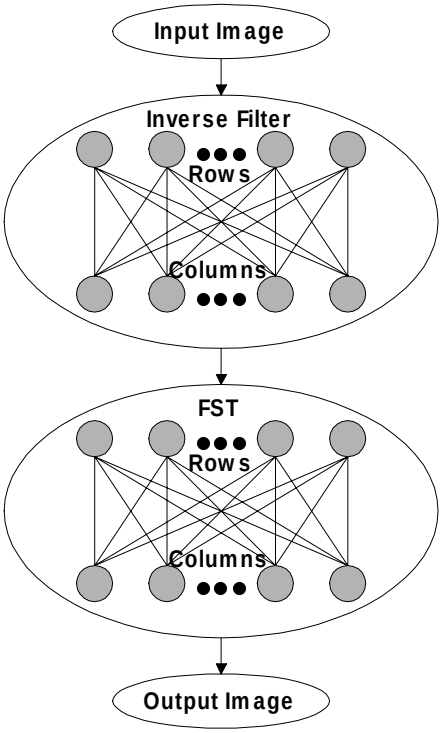

Figure 4-3: Computational Structure of Image Interpolation.

The diagram also exposes inherent parallelism in the computation. For both the IF and the FST, each row is independent of all the other rows and each column is independent of all the other columns. This allows simultaneous calculation of any number of rows or any number of columns.

4.2.2 Temporal Partitioning

Once the computational structure is understood, temporal partitioning can take place. The analysis of the computational structure indicates that the entire IF calculation must be performed before any of the FST calculation can be performed. The division between IF and the FST is an obvious candidate for a partition boundary. Within both the IF and the FST, no column can be computed until all of the rows have been computed. This is also a natural choice for a partition boundary. These decisions leave us with four partitions: IF rows, IF columns, FST rows and FST columns. Further partitioning might be necessary if the FPGA resources required by the computations in each of these partitions exceed those available on the platform. Alternatively, if the resources required represent a fraction of the available resources, then the required hardware may be duplicated within a partition to take advantage of the parallelism discovered in the computational structure. The exact choice of partitions and parallelism cannot be made until some implementation decisions have been reached.

4.3 The Application Interface

All applications conform to a Java interface that allows the tools to interface with the application in a well-defined manner. Listing 4-1 shows the application interface.

```
public interface Application {
      public void init(String args[]);
      public int getLogicRequirement();
      public int getMemorySizeRequirement();
      public int[] getValidMemoryWidthRequirement();
      public StageOrdered build();
}
```

Listing 4-1 : The Application Interface.

The `init(String args[])` method is called when the development tools are executing the application. It allows the application object to initialize its internal state and perform any other tasks that it might require. The image interpolator application object, for instance, creates a graphical user interface (GUI), loads an image into memory and converts it to gray scale.

The “get” functions that are defined next allow the tools to query the application to determine if it is capable of executing on a particular target platform. The prototype tool set allows the application programmer to require that the target architecture have certain properties. One is the minimum amount of configurable logic that must be available on a processing element. The `getLogicRequirement()` method returns this number. Another is the minimum size, in bytes, that the local memories on the CCM must have available. The `getMemorySizeRequirement()` method returns that number. Finally, the application programmer can require that the memories have a certain word size. A given application might provide support for multiple memory sizes so the `getValidMemoryWidthRequirement()` method returns an integer array containing all of the word widths that the application can support.

The `build()` method actually creates the ordered set of stages required by the application. The image interpolator’s `build()` method is shown in Listing 4-2.

```
public StageOrdered build() {
   // create all the stages
   StageOrdered interpolator = new StageOrdered();
   StageOrdered inv = new StageOrdered();
   StageOrdered fst = new StageOrdered();
   StageUnordered invRows = new StageUnordered();
   StageUnordered invCols = new StageUnordered();
   StageUnordered fstRows = new StageUnordered();
   StageUnordered fstCols = new StageUnordered();
   // inv rows
   for(int i=0; i<inputHeight; i++) {
      InvRowOp row = new InvRowOp(this,"inv row", i);
      invRows.Add(row);
   }
   // inv cols
   for(int i=0; i<inputWidth; i++) {
      InvColOp col = new InvColOp(this,"inv col", i);
      invCols.Add(col);
   }
   // fst rows
   for(int i=0; i<inputHeight; i++) {
      FSTRowOp row = new FSTRowOp(this,"fst row", i);
      fstRows.Add(row);
   }
   // fst cols
   for(int i=0; i<inputWidth*4; i++) {
      FSTColOp col = new FSTColOp(this,"fst col", i);
      fstCols.Add(col);
   }
   // hook stages together
   inv.Add(new StageSetup(gui, "IF Rows", inputHeight));
   inv.Add(invRows);
   inv.Add(new StageDone(gui, "IF Rows", inputWidth, inputHeight,
           invRowPixels));
   inv.Add(new StageSetup(gui, "IF Columns", inputWidth));
   inv.Add(invCols);
   inv.Add(new StageDone(gui, "IF Columns", inputWidth,
           inputHeight, invColPixels));
   fst.Add(new StageSetup(gui, "FST Rows", inputHeight));
   fst.Add(fstRows);
   fst.Add(new StageDone(gui, "FST Rows", inputWidth*4,
           inputHeight*4, fstRowPixels));
   fst.Add(new StageSetup(gui, "FST Columns", inputWidth*4));
   fst.Add(fstCols);
   fst.Add(new StageDone(gui, "FST Columns", inputWidth*4,
           inputHeight*4, fstColPixels));
   interpolator.Add(new AppSetup(gui, "Ready", 4));
   interpolator.Add(inv);
   interpolator.Add(fst);
   interpolator.Add(new AppSetup(gui, "Done", 4));
   return interpolator;
}
```

Listing 4-2: The Image Interpolator's build() Method.

The `build()` method in Listing 4-2 creates the representation of the image interpolator that was presented in Figure 3-5. It begins by creating three ordered stages that correspond to the image interpolator, the inverse filter and the fast-spline filter. Then four unordered stages are created that correspond to applying the inverse filter and fast-spline transform to the rows and columns of an image.

Next, the `build()` method fills each unordered stage with the operations that must be performed. For example, the first unordered stage that is filled corresponds to performing the inverse filter on the rows of an image. The `InvRowOp` operation performs this function. One `InvRowOp` is added to the `invRows` unordered stage for each row in the input image. This process is repeated for the next unordered stage by adding instances of the `InvColOp` operation to the `invCols` unordered stage. The process is performed twice more to create unordered stages representing the fast-spline transform on the rows and columns of the input image.

Having added all of the necessary unordered operations to the unordered stages, `build()` proceeds to order the unordered stages into the `inv` and `fst` ordered stage objects. Unlike unordered stages, the order that objects are added to ordered stages is important. This first thing that `build()` adds to the `inv` stage is something called a `StageSetup` object. `StageSetup` is an instance of `StageSoftware` object. Software stages are operations that are executed on the host instead of on the CCM. In this case, the `StageSetup` object updates the GUI.

Next, `build()` adds the `invCols` unordered stage which, as previously discussed, represents the inverse filter row operations. Finally, it adds another `StageSoftware` object called `StageDone`, that updates the GUI and performs some memory reordering tasks that are required to prepare for the next stage.

This process is then repeated again to add the operations necessary for the column operations. Then the `fst` ordered stage is built in a similar manner to contain the row and column operations for the fast-spline transform. Finally, the `inv` and `fst` ordered stages are added to the root level `interpolator` ordered stages along with some software stages that manage the GUI. The final `interpolator` stage is returned through the application interface.

4.4 Operations

Now that a cursory presentation of the overall structure of an application has been presented, a closer examination of the operations that make up an application is called for. For the purposes of the design automation tools, an operation is a self-contained unit that is capable of performing some useful computation. The operation object is responsible for generating its own I/O behavior using a standard memory interface specification defined by the tool framework. During a typical operation cycle, an operator is configured onto a target hardware architecture's processing element and connected to a memory interface. The operation's input data is moved from the host system to the memory that was assigned to the operation. The operator is given a signal to start

and it processes its data. It retires its results back into the memory to which it has been assigned. The host system retrieves the results from the FPGA's memory.

The interface definition for an Operation is given in Listing 4-3. The interface provides methods that are called by three different parts of the tool framework. The methods defined by the operation interface provide information to the compiler, describe how to implement the operation in hardware and how to perform communication between the application's software components and the operation's hardware instantiation.

```
public interface Operation extends Stage
{
      // Methods for the compiler
      public String getName();
      public int getLogicRequirement();
      public int getMemPortRequirement();

      // Method for building the hardware
      public abstract void buildOperation(Node parent,Wire
               data_in,Wire data_out,Wire addr,
               Wire write_sel,Wire strobe,Wire enable,
               Wire done);

      // Methods for the run-time system
      public MemoryImage userToCcm();
      public MemoryImage ccmToBuffer();
      public void bufferToUser(MemoryImage anImage);
}
```

Listing 4-3 : Operation Interface.

The first three methods allow the compiler to gather information about a particular operation. The `getName()` method returns an operation's name, which is used to uniquely distinguish the operation within the automated design tools. The `getLogicRequirement()` method allows the object to specify its logic requirements in terms of the quantity of configurable resources that it requires to build the operation on an FPGA. The `getMemPortRequirement()` method allows the object to specify the number of memory ports it requires to instantiate itself.

The next method, `buildOperation()`, describes the operation's hardware design. Its parameter interface represents the standard memory to which all operations must conform. The `Wire` type is a part of the JHDL design system that is used to describe an operation's hardware structure. More detail will be provided about JHDL and the structural description of an operator's hardware in a later section.

The last few methods define how an operation communicates between its hardware instantiation and the software application components. The `userToCcm()` method is used to load data from the application's memory space onto the FPGA processing platform. Listing 4-4 gives the method definition for `InvRowOp`'s `userToCcm()` method.

```
public MemoryImage userToCcm() {
      ImageInterpolator parent = this.getParent();
      int[] pixels = parent.getInputPixels();
      int w = parent.getInputWidth();
      return new MemoryImage(pixels, w*row, w, 0);
}
```

Listing 4-4: InvRowOp's userToCcm() Method.

The image interpolator application stores the data for the input image as an array of integers. The pixels are stored row by row. The job of `InvRowOp`'s `userToCcm()` method is to move one row of the input image from the host's memory to the accelerator board's memory. When the `InvRowOp` operation was created it was given a reference to the image interpolator application. The first thing that the `userToCcm()` method does is use this reference to retrieve a reference to the integer array containing the input image from the main application. It also queries the application for the width, in pixels, of a row in the image. Finally, the `userToCcm()` method sets up a `MemoryImage` object with the correct parameters to drive the actual memory copy process. This indirection is required since the operation's communication behavior must be defined without knowledge of the underlying hardware platform. Any particular hardware platform that has been targeted by the tools will contain a small bit of platform specific code that understands how to translate the contents of the `MemoryImage` object into the correct memory-copying behavior.

The constructor for the `MemoryImage` object takes as its first parameter a reference to an integer array containing the data to be copied to the hardware. The second and third parameters represent the index into the array to begin copying and the number of array elements to copy. The fourth parameter is the destination address on the processing array's memory to begin coping to. So, `InvRowOp`'s `userToCcm()` passes in the reference it got from the application to the image being processed, a start index based on the row that is being processed, the width of a row and a starting address to which to send the data.

The `ccmToBuffer()` method provides similar functionality for moving memory from the accelerator board to the host system. For performance reasons, the accelerator board's memory is not available in a random access fashion. Memory copied from the accelerator board must be done in a block beginning at some address and continuing for some known length. This seems the same as the behavior for `userToCcm()` but there are some subtle differences that require a more complex interface when moving data from the CCM back to the host.

```
public MemoryImage ccmToBuffer() {
      ImageInterpolator parent = this.getParent();
      int[] pixels = parent.getInvRowPixels();
      int w = parent.getInputWidth();
      return new MemoryImage(pixels, w*row, w, 0);
}
```

Listing 4-5: InvRowOp's ccmToBuffer().

Listing 4-5 shows the definition of `InvRowOp`'s `ccmToBuffer()`. It again pulls the reference to the application object and uses that to get a reference to a buffer that it can use to store the results. It also queries the width of a row. Then it sets up a `MemoryImage` object that describes the memory copying operation. This time, the host specific code will use to memory image object to set up a copy from the FPGA platform to the host. The CCM to host transfer, however differs somewhat from the host to CCM transfer. The asymmetry in the memory copying interface is due to the non-random access interface provided by the CCM. Listing 4-6 shows `InvColOp`'s `ccmToBuffer()`.

```
public MemoryImage userToCcm() {
      ImageInterpolator parent = this.getParent();
      int[] pixels = parent.getInvRowPixels();
      int w = parent.getInputWidth();
      int h = parent.getInputHeight();
      // build the col data
      int col_data[] = new int[h];
      for(int i=0; i<h; i++) {
            col_data[i] = pixels[i*w + col];
      }
      return new MemoryImage(col_data, 0, h, 0);
}
```

Listing 4-6: InvColOp's userToCcm() Method.

`InvColOp`'s `ccmToUser()` method is more complicated than `InvRowOp`'s `userToCcm()` method because the image is stored in memory column by column. When dealing with the row operations it is easy to just copy a linear block of memory from the hosts' memory onto the CCM's memory. When working with columns, the data must be pulled out of the input image and reordered into a temporary buffer so that it can be linearly copied onto the CCM. That is relatively simple to implement given the host's random access interface to its memory. It is more complex going the other way.

```
public MemoryImage ccmToBuffer() {
      ImageInterpolator parent = this.getParent();
      int h = parent.getInputHeight();
      int buffer[] = new int[h];
      return new MemoryImage(buffer, 0, h, 0);
}

public void bufferToUser(MemoryImage anImage) {
      int buffer[] = anImage.getData();
      ImageInterpolator parent = this.getParent();
      int[] pixels = parent.getInvColPixels();
      int w = parent.getInputWidth();
      int h = parent.getInputHeight();
      for(int i=0; i<h; i++) {pixels[i*w + col] = buffer[i];}
}
```

Listing 4-7: InvColOps's ccmToBuffer() and bufferToUser() Methods.

Listing 4-7 shows `InvColOp`'s `ccmToBuffer()` and `bufferToUser()` methods. The interface for reading data back from the FPGA platform is broken into two methods, because the run-time system must first retrieve the desired memory using a linear copy operation and then must potentially reorder it into the application's data space. This was simple in the case of `InvRowOp`. Since the application stores image information row by row, The `InvRowOp`'s `ccmToBuffer()` method simply set up the `MemoryImage` object to copy the results directly into the application's image buffer it did not require any reordering. Indeed, `InvRowOp`'s `bufferToUser()` method is an empty method. The `InvColOp`'s `ccmToBuffer()` method does not have this luxury. It must load the data into a temporary buffer, since the linear order that it reads memory from the hardware platform does not correspond to the order the results must be in the application's data space. Then `InvColOp`'s `bufferToUser()` method correctly reorders data from the temporary buffer into the application's image buffer.

```
else if ( curEvent instanceof FetchMemory ) {
      FetchMemory event = (FetchMemory)curEvent;
      Operation curOp = event.getOperation();
      for( int j=0 ; j<event.getNumMemPorts() ; j++ ) {
            MemoryImage mem = curOp.userToCcm();
            MemPair dest = event.getMemPortNum(j);
            this.LoadMemory(dest.pe,dest.mem,mem);
      }
}
else if ( curEvent instanceof RetireMemory ) {
      RetireMemory event = (RetireMemory)curEvent;
      Operation curOp = event.getOperation();
      for( int j=0 ; j<event.getNumMemPorts() ; j++ ) {
            MemPair dest = event.getMemPortNum(j);
            MemoryImage mem = curOp.ccmToBuffer();
            this.RetrieveMemory(dest.pe,dest.mem,mem);
            curOp.bufferToUser(mem);
      }
}
```

Listing 4-8: Code Fragment from the Run-Time System.

Listing 4-8 is a brief code fragment from the run-time system. Later, this will be covered in greater detail. At this point the import aspect of this code fragment is the manner in which the `Operation` interface is called.

`FetchMemory` events represent times when memory is being moved from the host to the accelerator. `RetireMemory` events are when data is being moved from the CCM to the host. The `LoadMemory()` method represents some architecture specific code that performs a memory copy operation from the host to the target architecture as described by a `MemoryImage` object. The `RetrieveMemory()` method performs an architecture specific operation to move data from the hardware platform to the host.

4.5 Building the Hardware with JHDL

As mentioned earlier, the actual structural description of the computational hardware is done using JHDL. JHDL hardware descriptions are architecture independent. JHDL designs are simply passed to a netlisting object that understands how to translate the JHDL structures into primitives on the target FPGA. Therefore, while a small amount of FPGA dependent code must be written to support a new platform, the JHDL designs themselves do not have to be modified. JHDL allows the user to describe hardware using an unmodified unextended high-level programming language, namely Java. In conjunction with the automated design framework presented here, this allows the user to describe their entire application using a single design representation.

The hardware design for the image interpolator is divided among several classes. This demonstrates the advantages of using Java's object-oriented inheritance mechanism for hardware design. The `ImageInterpolatorOp` class, which implements the `Operation` interface, is the base class for all of the `Operation` objects in the image interpolator. The `ImageInterpolatorOp` class implements some basic functionality common to all of the operations in the interpolator. It, for instance, implements the communications mechanism that allows instances of the operations to retrieve data from the main application objects.

Descended from the `ImageInterpolatorOp` operation are the `InvOp` and `FSTOp` operations. The hardware for both the inverse filter and the Fast-Spline Filter is the same whether it is operating on rows or columns, so some common functionality from the row and column operations can be represented in the `InvOp` and `FSTOp` classes. Since the hardware for both rows and columns is the same for the inverse and Fast-Spline filters respectively, the `InvOp` and `FSTOp` classes implement the `buildOperation()` method of the `Operation` interface.

The difference between row and column operations is represented at the level of the `InvRowOP`, `InvColOp`, `FSTRowOp` and `FSTColOp` classes. These classes implement the `userToCcm()`, `ccmToBuffer()` and `bufferToUser()` methods of the `Operation` interface. This allows the final layer of specialization to control how memory is transferred between the host and the hardware platform. The row operations copy memory to and from the host using linear buffers, while the column operations perform the reordering necessary to handle image columns.

The detailed JHDL descriptions of the filters themselves are contained in the `Filter` and `Tap` classes. These classes demonstrate the flexibility and convenience of using JHDL for representing structural hardware. Listing 4-9 is the core code that builds the FIR filters that appear in both the inverse and Fast-Spline Transform.

The code in Listing 4-9 creates a canonical pipelined FIR filter implemented using shift-and-add arithmetic. It follows the basic structure shown in Figure 4-4. The first few parameters to the method are references to JHDL `Wire` objects. These are used to connect up the hardware that the `build` method creates to higher-level hardware like the memory interface. The remaining parameters specify the kind of filter to be built.

```
private void build(Wire data_in, Wire acc_in, Wire enable,
                   Wire data_out, Wire acc_out,
                   int shift_values[][], boolean use_sv[][],
                   boolean negate[] )
{
      int width = data_in.getWidth();
      if( (acc_in.getWidth() != width) ||
            (data_out.getWidth() != width) ||
            (acc_out.getWidth() != width) )     {
            throw new BuildException(cellname + ": all I/O “ +
                                     “widths must be the same");
      }

      this.setGeneric("gw", width);
      port("data_in", data_in);
      port("acc_in", acc_in);
      port("enable", enable);
      port("data_out", data_out);
      port("acc_out", acc_out);

      Wire data = data_in;
      Wire acc = acc_in;
      Wire acc_out_int, data_out_int;

      for(int i=0; i<shift_values.length; i++) {
            if( i != shift_values.length-1 ) {
                  data_out_int = wire(width);
                  acc_out_int = wire(width);
            } else {
                  data_out_int = data_out;
                  acc_out_int = acc_out;
            }
            Tap tap = new Tap(this, data, acc, enable,
                              data_out_int, acc_out_int,
                              shift_values[i][0], use_sv[i][0],
                              shift_values[i][1], use_sv[i][1],
                              negate[i],
                              (new Integer(i)).toString());
            this.setDataDelay(this.getDataDelay() +
                              tap.getDataDelay());
            this.setAccumulatorDelay(this.getAccumulatorDelay() +
                                     tap.getAccumulatorDelay());

            data = data_out_int;
            acc = acc_out_int;
      }
}
```

Listing 4-9 : Core JHDL Design for a Filter.

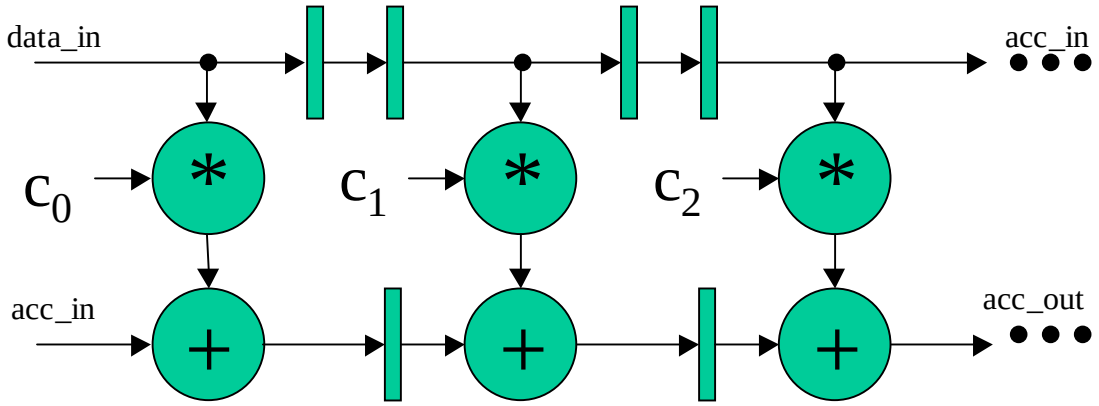

Figure 4-4: Generalized Filter Tap.

The `build` method begins by performing some consistency checking on its inputs. All the input wires should be references to previously created `Wire` objects. In addition, they should all have the same bit width. If the `build` method discovers inconsistencies, it throws an exception. The `build` method completes its housekeeping chores by creating its ports using the `port` method.

After finishing its housekeeping, the `build` method constructs the hardware required to perform the filtering operation. The `shift_values` array passed into the `build` operation specifies the coefficients for the filter. Recall that the coefficients of the filters in the image interpolator can all be implemented with simple shift and add structures. Most of the coefficients require only a single shift, but some require two shifters, so `shift_values` is a rectangular two-dimensional array. The first dimension corresponds to the length of the filter, i.e., the number of taps. The second index selects whether the addressed integer element of the array specifies the shift value for the first or second possible shifter. The `use_sv` array is a similarly sized array of boolean values. The indexes are analogous. The boolean elements indicate whether the specified shifter is used or not. If a shifter in a given tap is not needed, the `build` method will not create one. The last parameter passed into the `build` method is a one-dimensional array of booleans that indicate whether the coefficient is negative or not. If the coefficient is negative, the `build` method instantiates a negation unit in the pipeline. Since each tap could have a different delay depending on the parameters passed into it, the `build` method also keeps track of the total delay through the filter pipeline. This information is important for the FST filter units, since they involve two FIR filters whose delays must be matched.

The main portion of the `build` method's construction operation takes place within the `for` loop. The `build` method knows how many taps there are in the filter by querying the length of the `shift_values` array. It iterates through the loop once for each filter tap and calls the `Tap` class constructor with parameters characterizing each individual tap. The `Tap` constructor instantiates a filter tap based on the parameters passed to it.

The tap constructor does some initial housekeeping, which was deleted to save space, and then constructs a tap that falls into one of the three categories diagrammed in Figure 4-5. The "<>" operators represent shifters.

```
public Tap(Node parent, Wire data_in, Wire acc_in, Wire enable,
           Wire data_out, Wire acc_out,
           int shift_bits_0, boolean do_shift_0,
           int shift_bits_1, boolean do_shift_1,
           boolean negate, String name)
{
   super(parent, cellname+name);

   // Housekeeping stuff removed for brevity...

   // delay data_in once more when double shifting
   Wire data_in_del = data_in;
   Wire acc_in_del = acc_in;

   if( !do_shift_0 && !do_shift_1 ) {
      this.setDataDelay(2);
      this.setAccumulatorDelay(1);
      regce_o(acc_in_del,enable,acc_out);
   } else {
      Wire tap;
      if( do_shift_0 && !do_shift_1 ) {
         tap = this.getTap(data_in_del, shift_bits_0);
         this.setDataDelay(2);
         this.setAccumulatorDelay(1);
      } else if( !do_shift_0 && do_shift_1 ) {
         tap = this.getTap(data_in_del, shift_bits_1);
         this.setDataDelay(2);
         this.setAccumulatorDelay(1);
      } else { // shift both
         Wire tap_0, tap_1;
         tap_0 = this.getTap(data_in, shift_bits_0);
         tap_1 = this.getTap(data_in, shift_bits_1);
         tap = regce(add(tap_0,tap_1),enable);
         data_in_del = regc(data_in_del);
         acc_in_del  = regc(acc_in_del);
         this.setDataDelay(3);
         this.setAccumulatorDelay(2);
      }
      regce_o(addsub(acc_in_del,tap,constant(1,(negate?0:1))),
                     enable,acc_out);
   }

   regce_o(regce(data_in_del,enable),enable,data_out);
}
```

Listing 4-10 : Core JHDL Design for a Filter Tap.

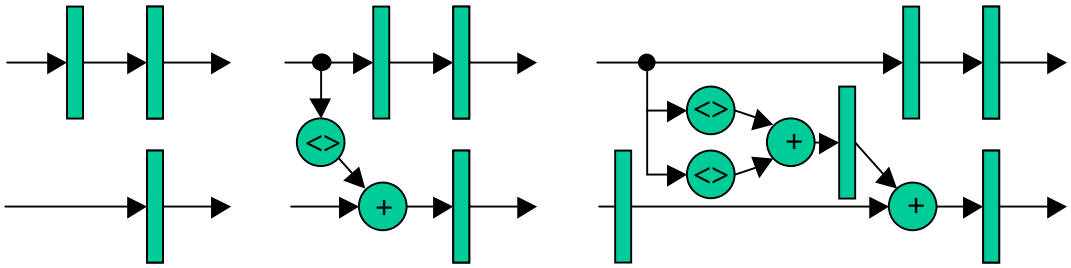

Figure 4-5 : Possible Filter Taps.

Which of the three possible filter taps is created is dependent on the arguments passed into the `Tap` constructor. The actual calls to the JDHL `Logic` library that instantiate the low level shifters can be found in the `Tap` class's `getTap()` method. Listing 4-11 shows the code for the `getTap()` method.

```
private Wire getTap(Wire data_in, int shift_bits) {
      Wire tap;

      if( shift_bits > 0 ) {
            tap = shiftl(data_in, shift_bits);
      } else if( shift_bits < 0 ) {
            tap = ashiftr(data_in, -shift_bits);
      } else { // sh == 0
            tap = data_in;
      }

      return tap;
}
```

Listing 4-11 : Tap's getTap() Method.

Chapter 5 Architecturally Retargetable Compilation

The previous chapter discussed the manner in which applications are described to the automated design tools. This chapter examines how that description is compiled into a RTR application for a target architecture. The chapter begins with an examination of the hardware abstraction that is used to describe the target architecture. A discussion of the run-time system follows. Finally, the compilation system itself is presented.

5.1 The Hardware Abstraction

Architecturally independent RTR CCM applications are written to execute on an abstracted hardware platform that is represented by the `HardwareArchitecture` class. A pseudo-class description of the `HardwareArchitecture` class is shown in Listing 4-1.

```
public abstract class HardwareArchitecture
{
      // Processing element interface for building hardware
      private Hashtable elements;
      protected void Add(ProcessingElement p);

      // Compile-time hardware information interface
      public int getNumElements();
      public int getMaxLogic(String n);
      public int getRemLogic(String n);
      public int getMaxMemPort(String n);
      public int getRemMemPort(String n);

      // Compile-time hardware assignment interface
      public Enumeration getElements();
      public void addToElement(String s,Operation o);
      public Vector getHardOp(String pe,String op);
      public Hashtable getAllHardOps(String pe);
      public void ClearHardOps();

      // Compile-time configuration generation interface
      public void createBitFile(Configuration c);
```

```
	// Run-time hardware interface
	public abstract void LoadMemory(String pe,String mem,
	                                MemoryImage image);
	public abstract void RetrieveMemory(String pe,String mem,
	                                    MemoryImage image);
	public abstract void LoadConfiguration(Configuration c);
	public abstract void ExecuteHardware();
}
```

Listing 5-1 : `HardwareArchitecture` Pseudo-Class Definition.

The `HardwareArchitecture` class is an abstract class that defines three basic interface components. The first is an interface called by inheritors of the `HarwareArchitecture` class to create processing elements within the architecture. The second is an interface used by the compiler to query information about the architecture, assign operations to processing elements in the architecture, and generate configurations for processing elements based on the operators assigned to them. The third interface is used by the run-time system to perform abstract operations on the target architecture like loading memories and configuring processing elements.

There is also an abstract base class that defines a closely related abstraction for processing elements. When a concrete class inherits from the `HardwareArchitecture`, it creates processing elements for itself that are instances of a concrete implementation of the `ProcessingElement` class. A pseudo-class definition for the `ProcessingElement` class is shown in Listing 5-2.

```
	public abstract class ProcessingElement {
		// Constructor
		public ProcessingElement(String n);
		// Name interface
		private String name;
		public String getName();
		// Information interface
		private int LogicRemaining;
		private int MemPortRemaining;
		public int getRemLogic();
		public int getRemMemPort();
		public abstract int getMaxLogic();
		public abstract int getMaxMemPort();
		// Operation assignment interface
		private Hashtable ops;
		private Stack MemPortNamesRemaining;
		public void AddOperation(Operation o);
		public String getUnusedMemPort();
		public Vector getHardOp(String s);
		public Hashtable getAllHardOps();
		public void ClearHardOps();

		// Configuration creation interface
		public abstract void CreateBitFile(Configuration c);
	}
```

Listing 5-2: `ProcessingElement` Pseudo-Class Definition.

The `ProcessingElement` class provides interface functions that allow the `HardwareArchitecture` class to query information about the processing elements that it contains. It provides an interface that allows the `HardwareArchitecture` object to assign operations to its various processing elements. The `ProcessingElement` architecture also provides an abstraction for an architecture dependent process that creates hardware configurations for the operations assigned to a processing element.

To fully understand the hardware abstraction, it needs to be examined in the context of a concrete implementation. The `WildForceXL_4062` class inherits from the `HardwareArchitecture` class and represents all of the hardware specific code required to target the automated tools to an architecture. The constructor for the `WildForceXL_4062` class is shown in Listing 5-3.

```
public WildForceXL_4062()
{
      this.OpenBoard();

      super.Add(new W4_XC4062("pe1"));
      super.Add(new W4_XC4062("pe2"));
      super.Add(new W4_XC4062("pe3"));
      super.Add(new W4_XC4062("pe4"));
}
```

Listing 5-3: `WildForceXL_4062` Constructor.

The constructor begins by calling a private method of the `WildForceXL_4062` class that initializes the hardware platform. The `OpenBoard()` method is implemented as a native method and calls the appropriate functions from the WildForce API. Next, the constructor creates instances of the `W4_XC4062` class to represent its processing elements. The `W4_XC4062` class is an inheritor of the `ProcessingElement` class. An abbreviated class definition for `W4_XC4062` is given in Listing 5-4.

```
public class W4_XC4036 extends ProcessingElement {
      public W4_XC4036(String n) { super(n); }
      protected final int MaxLogic = 3138;
      protected final int MaxMemPort = 1;
      public int getMaxLogic() { return MaxLogic; }
      public int getMaxMemPort() { return MaxMemPort; }
      public Stack getMemPorts() {
            Stack v = new Stack();
            v.push("mem0");
            return v;
      }
      public void CreateBitFile(Configuration c) {
            // Code for producing XC4062 netlist and
            //  processing it to a bit file.
      }
}
```

Listing 5-4: Abbreviated `W4_XC4062` Class Definition.

The W4_XC4062 class provides some constants that indicate the logic capacity of the FPGA and the number of memory ports available. The WildForce processing elements provide a single memory interface per processing element. The getMemPorts() method's contract is to provide a stack of strings that identify the memory ports available on the processing element. Since WildForce processing elements provide a single memory interface, the stack returned contains a single string, "mem0". If the processing element supported multiple memory interfaces there would be multiple strings. Clients of the W4_XC4062 class use these strings to uniquely identify memory ports on the processing element.

The other piece of functionality that the W4_XC4062 class brings to the ProcessingElement hierarchy is the architecture dependent code for generating a hardware configuration. For Xilinx FPGAs these configurations are commonly referred to as bit files.

So, returning to Listing 5-3, the WildForceXL_4062 constructor creates four W4_XC4062 objects to represent its processing elements. The string parameter passed to the W4_XC4062 object specifies the name that the HardwareArchitecture and its clients will use to refer to the processing element.

The other functionality that WildForceXL_4062 extends from HardwareArchitecture are implementations of the abstract hardware operations: LoadMemory, RetrieveMemory, LoadConfiguration and ExecuteHardware. These functions are all implemented as native methods that make appropriate calls to the WildForce API. The finalizer for the WildForceXL_4062 class calls a private native method called CloseBoard that closes the WildForce board.

Referring back to Listing 5-1, enough information has now been presented to further explain the HardwareArchitecture's basic interface. The elements hash table and the Add() interface are used by concrete inheritors to create and name an architecture's processing elements.

The compile-time information interface allows clients of the HardwareArchitecture class to query information about the hardware architecture itself and constituent processing elements. The functions that deal with processing elements take a string as a parameter. The string is the name of the processing element as defined by the concrete architecture's constructor. The names themselves can be retrieved using portions of the hardware assignment interface.

The compile-time hardware assignment interface is used by the design compiler to assign operations to specific processing elements. The getElements() method will return an enumeration of the processing elements in the hardware architecture. The addToElement() can be used to assign an Operation to a specific processing element. The processing element will, in turn, automatically assign the Operation to one of its available memory ports.

Some of the information interface methods make more sense in this context. The getMax…() style methods return the ultimate capacities of the specified processing element. The getRem…()

style methods return the remaining resources on a processing element given the operations currently assigned to it.

The …HardOp() style methods are used to interface with operations that have already been assigned to processing elements. The getHardOp() method will return a vector of data structures that contain information about a specific type of operation assigned to a specific processing element. If, for instance, processing element "pe1" had an InvRowOp operation assigned to it, a call to getHardOp("pe1","InvRowOp") would return a vector containing information about all of the InvRowOp operations that have been assigned to "pe1". The information includes a reference to an instance of the operation and the memory port assigned to each instance of the operation assigned to that processing element.

The compile-time interface for generating hardware configurations consists of a single method called createBitFile() that takes a Configuration object as a parameter. A pseudo-class definition for the Configuration object is given in Listing 5-5. A Configuration object hardware configuration for the a processing element on the RTR CCM. It is a simple data structure that stores the name of the processing element, the name of the bit file to be loaded onto the processing element, and a hashtable that contains information about the Operation objects assigned to the procesing element. The createBitFile() method examines the processing element field of the Configuration passed to it and passes it on to the createBitFile() method of the appropriate processing element. The overall contract of the method is to produce a hardware configuration for the designated element.

The run-time hardware interface provides an abstraction of the underlying platform that is common to any architecture capable of supporting applications produced by the automated design tools. The LoadConfiguration() method takes a Configuration object as an input and loads the appropriate bit file onto the specified processing element. The LoadMemory() and RetrieveMemory() methods take a processing element name, a memory name and a MemoryImage object as parameters and load or retrieve the specified buffer to or from the specified memory on the indicated element. The ExecuteHardware() method causes the hardware to execute until the current set of Operation objects have been completed.

```
public class Configuration {
      private String pe;
      private String bitfile;
      private Hashtable ops;
      // Constructor
      public Configuration(String pe,String className,String
                           bitfile,Hashtable ops);
      // Data retrieval methods
      public String getProcessingElement();
      public String getBitFile();
      public Hashtable getOps();
}
```

Listing 5-5: `Configuration` Pseudo-Class Definition.

With a firmer understanding of the underlying hardware abstraction, enough information has been presented that some of the details of compilation process can be examined.

5.2 The `RunTime` Class

The `RunTime` class implements the basic run-time functionality of an application. Recall the structure of an application. An application consists of a hierarchical tree of ordered and unordered stages. Using a defined set of ordering rules, the hierarchical structure can be flattened into a unique ordered sequence of `StageUnordered` objects. Each `StageUnordered` object contains a set of `Operation` objects. Each `Operation` object defines methods for moving data back and forth between the host and the RTR CCM. Each operation also provides a description of a hardware operation that reads data from a memory interface, performs computations, and returns results back to the memory interface. No restrictions are placed on the hardware defined for the memory interface. It may read and write data in any pattern that it wishes. The `Operation` objects all target an abstract memory interface. Since memory interfaces differ in their sizes and word widths, the application programmer is allowed to specify a set of word widths that their application will support and a minimum size for a memory on a potential platform. An application can also perform some of its computation or housekeeping operations in software by including `StageSoftware` objects in the hierarchy.

At run time, the operations need to be sequenced onto the RTR CCM. This occurs as the execution of a series of `RunTimeEvent` objects created by the compiler. All user applications are reduced to a series of these event objects. The process that generates this sequence will be discussed in Section 5.3. For now, there are five types of run-time events. They are all inheritors of the `RunTimeEvent` abstract class. The `RunTimeEvent` hierarchy is diagrammed in Figure 3-8.

The code fragment that implements the run-time execution of the `ConfigureProcessingElements` is given in Listing 5-6. The `ConfigureProcessingElements` event causes hardware configurations to be loaded onto the target platform. The event has an array of `Configuration` objects in it that contain the information about which processing element should be programmed with which bit file. Execution of the configuration event involves simply cycling through the array of configurations and calling the hardware architecture's `LoadConfiguration()` on each one.

```
if( curEvent instanceof ConfigureProcessingElements ) {
   ConfigureProcessingElements event =
                                ConfigureProcessingElements)curEvent;

   for( int j=0 ; j<event.getNumConfigs() ; j++ )
   {
      hardware.LoadConfiguration(event.getConfigNum(j));
   }
}
```

Listing 5-6: `ConfigureProcessingElements` Code Fragment.

The FetchMemory event fetches memory from the host-side application's data space and writes it into a memory on the accelerator platform. There is a fetch memory event associated with the execution of every operation. The event causes the data that the operation needs to be loaded into the proper memory for processing. The FetchMemory object has a reference to the operation that is to be executed. It calls the operation's userToCCM() method to get an array of MemoryImage objects that need to be moved to the processing array. Each element of the MemoryImage array is passed to the hardware architecture's LoadMemory() method. A code fragment demonstrating the actions taken to resolve a FetchMemory event is shown in

```
else if ( curEvent instanceof FetchMemory ) {
   FetchMemory event = (FetchMemory)curEvent;
   Operation curOp = event.getOperation();
   for( int j=0 ; j<event.getNumMemPorts() ; j++ )
   {
      MemoryImage mem[] = curOp.userToCcm();
      MemPair dest = event.getMemPortNum(j);
      for( int memNum=0 ; memNum<mem.length ; memNum++ )
      {
         hardware.LoadMemory(dest.pe,dest.mem,mem[memNum]);
      }
   }
}
```

Listing 5-7: `FetchMemory` Code Fragment.

The RetireMemory event represents the complementary operation to the FetchMemory event. RetireMemory causes the processed data from an executed operation to be brought back from the CCM and returned to the application's data space. RetireMemory is more complex than FetchMemory because of the random memory access issues discussed in Section 4.4. The fetch event has a reference to the operation that has been executed and calls its ccmToBuffer() method to get an array of MemoryImage objects that need to be moved from the CCM to the host. It calls the hardware architecture's RetrieveMemory() method for each MemoryImage object in the array. As demonstrated by the example shown in Listing 4-7, the application code can set up the MemoryImage objects returned by ccmToBuffer() to reference temporary buffers. These same MemoryImage objects are then passed to the operation's bufferToUser() method, so that the application code can replace the memory in any desired fashion. Again, as discussed in Section 4.4, if this step is unnecessary, the application code can simply define an empty method for bufferToUser(). A code fragment showing the process invoked by a RetireMemory object is shown in Listing 5-8.

Both of the interlude events have relatively simple implementations. The HardwareInterlude object represents the hardware execution of an operation. It simply calls the ExecuteHardware() method of the hardware architecture. The SoftwareInterlude object represents the execution of a StageSoftware object. It has a reference to the StageSoftware object to be executed and simply calls its execute() method. A code fragment showing the behavior of the two interlude events can be found in Listing 5-9.

```
else if ( curEvent instanceof RetireMemory ) {
   RetireMemory event = (RetireMemory)curEvent;
   Operation curOp = event.getOperation();
   for( int j=0 ; j<event.getNumMemPorts() ; j++ )
   {
      MemPair dest = event.getMemPortNum(j);
      MemoryImage mem[] = curOp.ccmToBuffer();
      for( int memNum=0 ; memNum<mem.length ; memNum++ )
      {
         hardware.RetrieveMemory(dest.pe,dest.mem,mem[memNum]);
      }
      curOp.bufferToUser(mem);
   }
}
```

Listing 5-8: `RetireMemory` Code Fragment.

```
else if ( curEvent instanceof HardwareInterlude )
{
   hardware.ExecuteHardware();
}
else if ( curEvent instanceof SoftwareInterlude )
{
   SoftwareInterlude event = (SoftwareInterlude)curEvent;
   event.execute();
}
```

Listing 5-9: Interlude Event Code Fragment.

At run time, applications follow a basic cycle of these events to execute an application. Applications begin by executing a series of `ConfigureProcessingElements` events to load hardware configurations onto the processing elements on the RTR CCM. Next, input data is moved to the processing elements' local memories by `FetchMemory` events. Then a `HardwareInterlude` event causes the hardware to execute the currently configured operations. When that has finished, `RetireMemory` events move the resultant data from the accelerator back to the host. The exact series of events and the repetition of the cycle are determined by the compiler.

5.3 The `Compiler` Class

The `Compiler` class is responsible for taking the application designer's description of their application and converting it into a set of configurations for the hardware platform and a series of run-time events for the run-time system to carry out. The prototype compiler, called `BasicCompiler`, makes some assumptions about the characteristics of the applications that it will compile. These assumptions result in simplifications to the compilation process and are not requirements of the overall automated design strategy. More advanced compilers could certainly be written that allow these initial set of assumptions to be relaxed. The `Compiler` abstract class provides a fixed interface so that future compilers can be seamlessly inserted into the tool set.

The primary limitation of the `BasicCompiler` is that is can only handle `UnorderedStage` objects that contain a single type of operation. Even given this restriction, many different kinds of interesting applications can be developed. The image interpolator, for instance, does not suffer at all from this restriction. The `invRows` unordered stage contains only one type of operation, the `InvRowOp` operation. Unordered stages can contain many instances of that operation. The `invRows` stage contains one `InvRowOp` per row in the input image. Under the restriction, `invRows` could not, for instance, contain both `InvRowOp`'s and `InvColOp`'s.

The `BasicCompiler` begins by insuring that the application description passed to it meets the single operation type requirement. The compiler's outputs will be a `RunTimeScript` object and a set of EDIF netlists for hardware configurations.

The compiler works on one unordered stage at a time. The first step is to design hardware for executing the application. The compiler examines the stage and determines what type of operation is used in it. Queries are made to the target hardware architecture to determine how many processing elements are available, the sizes of the elements, and the number of memory interfaces that each element supports. The compiler then determines how many instances of the operation can be built on the RTR CCM subject to the restrictions on the number of available memory ports and the amount of configurable logic available per device. Given that each operation requires a memory port, in the absence of some other limiting constraint, one operation can be instantiated per memory port available in the processing element. Sometimes, this is not possible due to restrictions on the amount of configurable logic that is available. Even if a given processing element has multiple memory interfaces, if an operation consumes enough of the element's configurable resources, it may not be possible to assign multiple instances of the operation to that element.

Once the compiler has maximized the number of operations that it can instantiate in hardware, it calls the JHDL technology mapper to produce netlists for programming the processing elements. Methods from the `Operation` objects are called to generate the operation specific hardware. Concrete implementations of the hardware architecture abstraction provide any platform specific interface hardware that is necessary. The result is a series of netlists that are processed through vendor tools to produce hardware configurations.

At this point, the compiler has created hardware instantiations of the operator type for this stage on the platform's processing elements. Each hardware instantiation of the operation is capable of executing any of the specific instances of the operations represented in the unordered stage. In the example of the image interpolator targeted for the WildForce board, the compiler creates one hardware instance of the `RowInvOp` operation on each of the WildForce's processing elements. The `invRows` stage, for a 128x128 image, has 128 instances of the `RowInvOp` operator in it. So the compiler's next step is to schedule each of the 128 `RowInvOp` operations on one of the hardware instances.

The compiler accomplishes this by adding a series of events to the `RunTimeScript` object. The first event is a `ConfigureProcessingElements` event that causes the hardware configurations,

generated by the netlists produced earlier, to be loaded onto the processing elements. In the interpolator on WildForce example, four Xilinx bit files are loaded onto the four processing elements. Next, the compiler chooses one instance of the operation from unordered stage for each available hardware instantiation. In case under discussion, the compiler will choose four `RowInvOp` operations from the `invRows` stage and assign each of them to one of the operation instantiations on the four processing elements. `FetchMemory` events are added to the `RunTimeScript` for each of the operation instances selected to move their input data from the user's memory space to the local memory space of the processing elements. A `HardwareInterlude` event is then added to the `RunTimeScript` to execute the hardware configuration. The processing elements perform the operations assigned to them and store their results in the processing element's memory space. So, the compiler adds a set of `RetireMemory` events that move the resulting data from the accelerator board back to the user's memory space.

One of two things can happen next. If all of the operation instances from the unordered stage have been scheduled on the hardware, the compiler proceeds to the next unordered stage. If operations remain, the cycle discussed above is repeated with one exception. Since hardware configured to perform the desired set of operations is already present on the RTR CCM, no `ConfigureProcessingElements` event is generated. The compiler simply generates another set of fetch, execute and retire events.

If the application contains `StageSoftware` objects, the compiler will translate those into `SoftwareInterlude` events.

When the entire process is finished, the compiler has produced a series of run-time events that will sequence the user's application on the target hardware.

Chapter 6 Application Development

Previous chapters have examined elements of the application development process, but they have done so within the context of explaining other aspects of the Janus tool design. This chapter outlines the application development process in its own right. The first section explains the basics of the design flow and the following sections present brief case studies of some of the applications implemented as part of the Janus research.

6.1 Design Flow

This section examines the application development process supported by the Janus tools. The development process begins with a specification of the application to be implemented. The developer analyzes the specification and develops a partitioning of the overall application process into stages. Each stage is further decomposed into a set of operations that can be implemented in hardware. The operations themselves are then implemented and organized into a sequenced set of ordered and unordered stages. This conglomeration can then be automatically optimized for a target CCM.

6.1.1 Initial Design Partitioning

The development process begins with the initial design specification. From the specification, a Janus developer derives the basic computational dependencies of the application's computational flow. For applications that are appropriate targets for the Janus tools, these dependencies tend to describe natural boundaries for the creation of computational stages. In the case of the image interpolator, these stages correspond to performing an inverse filter on an image's rows, an inverse filter on an image's columns, a Fast Spline Transform on an image's rows, and a Fast Spline Transform on an image's columns. At this point, the developer can create a basic framework for the application, based on the initial partitioning. This decomposition corresponds exactly with the structural description found in the image interpolator application's build method from Listing 4-2.

6.1.2 Software Implementation

A Janus developer could immediately begin to investigate hardware implementations of the various stages, but a more prudent approach might be to initially build a software implementation of the application. This is readily done within the Janus development system.

The existence of software stages allows the developer to build a complete model of the computation system within the same structural framework that the hardware will be implemented. A complete software model of the system allows the developer to concentrate on refining the algorithms and methods that the hardware will eventually implement. During the development of the stereo imaging application described in Section 6.3, several different algorithms for performing the image matching operation were investigated. The implementation of the differing algorithms within the software stage framework made it easy to swap different algorithms in and out of the computational stream while evaluating the quality of the results. Exploration of the design space is much easier in software than in hardware. The result of the exploration, in the case of the stereo matching algorithm, was the selection of a method that had a much simpler hardware implementation that the method initially envisioned.

Development of the software model also has collateral benefits. In the process of developing it, all of the effort required to perform translation of data from external file formats or to retire results in particular formats is reusable for the hardware application. This is true of any graphical user interface elements developed as well. Another significant benefit of the software model is the ability to perform incremental conversion to hardware.

6.1.3 Incremental Hardware Development

With a working software stage model of the application, the developer can incrementally replace the software stages with hardware stages. Any software stage in the end-to-end computational model can be chosen for conversion. During the development of the image interpolator, the hardware stages where built in the same order that they are sequenced at run time, but this is not a requirement. If the design of one stage of the hardware is critical to the design of the other stages, that element may be developed and simulated using the existing software model in place of the other computational stages.

The selected stage is analyzed and then broken down into operations. In the case of the image interpolator, suppose the inverse filter computation on the rows of the image is selected. The inverse filter row computation is made up of *n* FIR filter computations, where *n* is the number of rows in the input image. Each FIR filter computation is can be calculated independently of all of the other FIR filter operations. This characteristic lends itself to parallel implementation. At the next level of granularity, each FIR filter computation is made up of many interdependent multiply accumulates. The FIR filter computation cannot be easily made parallel. It can, however, be easily pipelined. Close examination of the structure of the FIR filter also reveals that further division of the computation would result in operations that would require that each row of the input image to be read from memory more than once. If the FIR filter computation is chosen as the basic operation for this stage, each row of the input image only needs to be read from memory and passed through an operation once. These characteristics would lead a designer to choose to make each FIR filter computation an operation.

The next stage of an unaided design flow might be to examine how many of these operations could be implemented on the target platform. The Janus tools, however, remove this requirement. The developer only needs to implement a single operation object and the compiler

will determine how to implement parallel instances of the operation on the target hardware. If the target hardware is capable of supporting several instances of the operation, the compiler will generate them. If the hardware is capable of supporting only a single operation, the compiler will generate that. The design interface for operation objects makes it irrelevant to the developer how many of the operations will be instantiated at run time.

Another advantage of the retargetable nature of the Janus tools is evident during simulation. The Janus tools come with a simulated hardware architecture specifically for development purposes. This hardware architecture provides the same functional interface to the operation object that a real hardware architecture would. This allows testing of operations to be conducted without generating netlists and going through the place-and-route procedure that would be required to execute on hardware. The simulated hardware architecture provides a fully functional interface to the Janus run-time system as well. This allows the testing process to the same user interface that will be used when the application is running in hardware.

When a hardware instance of the operation has been debugged, its results can be automatically compared with the results produced by the corresponding software stage. This sort of testing is possible even if the software stage is not bit-accurate to the hardware implementation. Suppose that the software stage uses floating-point math to calculate its result while the hardware version uses fixed-point. The data resulting from the hardware operation is available to the host-side Java run-time environment. With the complete features of the Java programming language at their disposal, developers are free to perform whatever conversions are required to make comparisons between results.

The software implementation also provides a built-in benchmark for testing the speed of the hardware implementation. If the software stage is carefully implemented it can provide immediate feedback about the hardware's speedup over a microprocessor-based implementation. Since software stages are fully functional implementations of computational stages, if the hardware implementation fails to exceed the performance of the software implementation, the software implementation can be used in the final version of the application.

The ability to mix software and hardware stages together also allows designers to use the host to perform operations, like data reordering, that are not well-suited for implementation on FPGAs.

6.1.4 Application Compilation

The Janus Control Center's graphical user interface is divided into four panes. When a design description is loaded into the Control Center, using the **Application** menu, the design hierarchy is shown in a tree structure in the upper left pane. This tree structure corresponds to the composition of **Stage** objects assembled by the developer to specify the application. The Control Center in Figure 6-1 has been loaded with the design description for the image interpolator. The various ordered and unordered stages are labeled in the diagram. The first two inverse filters, labeled IF Rows and IF Columns in the figure, are visible in the design pane. When a stage is selected in the upper left pane, the lower left pane displays information about that stage.

After loading an application, the user specifies a hardware architecture using the **Platform** menu. The compiler is then invoked from the **Build** menu. When the compiler has finished, it displays the series of **RunTimeEvents** that it produced in the upper right pane of the Control Center. The initial sequence of run-time events for the image interpolator is visible in Figure 6-1. They follow the general computational cycle of Janus applications discussed in Section 3.2. Some extra software interludes are visible in Figure 6-1. These stages interact with the image interpolator's graphical user interface to update status and provide the user with views of intermediate data. When an event is selected in the upper right pane, information about it is displayed in the lower right pane.

6.1.5 Application Execution

Execution of the application is invoked using the **Control** menu. The user can cause the entire series of run-time events to be played from beginning to end by selecting the **Play** option. Alternatively, the user can proceed through the application one run-time event at a time using the single step options. While the application is executing the application's own graphical user interface is the primary point of interaction with the user.

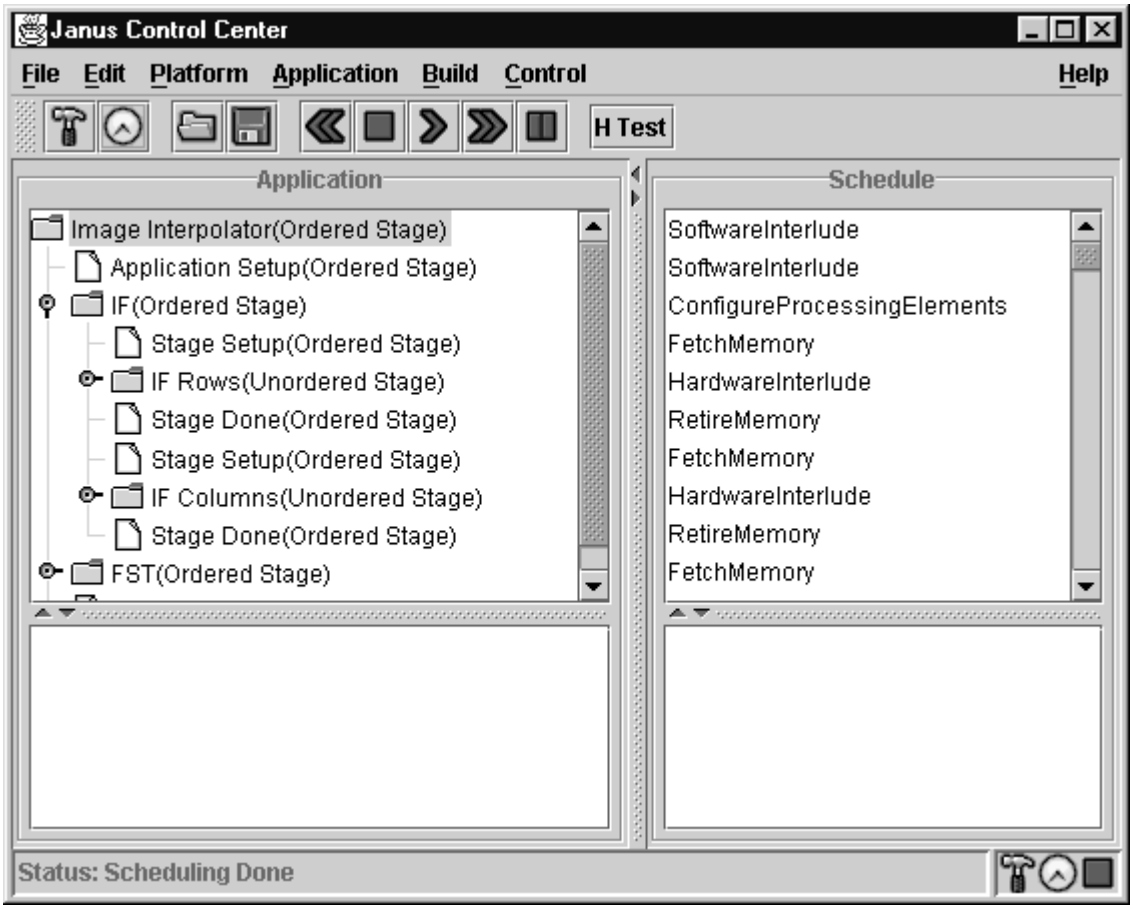

Figure 6-1: Janus Control Center.

6.2 Image Interpolator

The image interpolator's graphical user interface is shown in Figure 6-2. It has a **File** menu that allows the user to select an input image for interpolation and a **Test** menu that allows selection of various test patterns. Some image zoom controls are provided to allow detailed examination of image features. The interface has two progress bars that monitor report the status toward completion of individual stages and the overall process respectively. The lower portion of the interface displays the initial image, output image and, optionally, intermediate images produced between each stage.

Figure 6-2: Graphical User Interface for the Image Interpolator.

6.3 Stereo Depth Finding

The stereo depth finding application is another example of a multi-stage computation built using the Janus tools. The stereo depth finder processes two input images produced from two slightly displaced points of view. The displacement of image features within the two images can be used to extract depth of view information. This implementation of the stereo depth finding algorithm is a three-stage process. The first stage is to perform edge detection on the input images. This emphasizes high-frequency portions of the image that are typically good candidates for selection as significant image features. The next stage is thresholding of the edge-detected images. This removes high-frequency noise from the image and improves results of the matching stage, which follows. The matching stage examines each pixel that has survived the previous thresholding stage and attempts to match it with a pixel in the other image. If the matching process is successful, then the matched pixels correspond to the same image feature as seen from two slightly different points of view. The difference in the x-y location of the pixel from one image to the other, can be used to extract information about the z coordinate of the image feature.

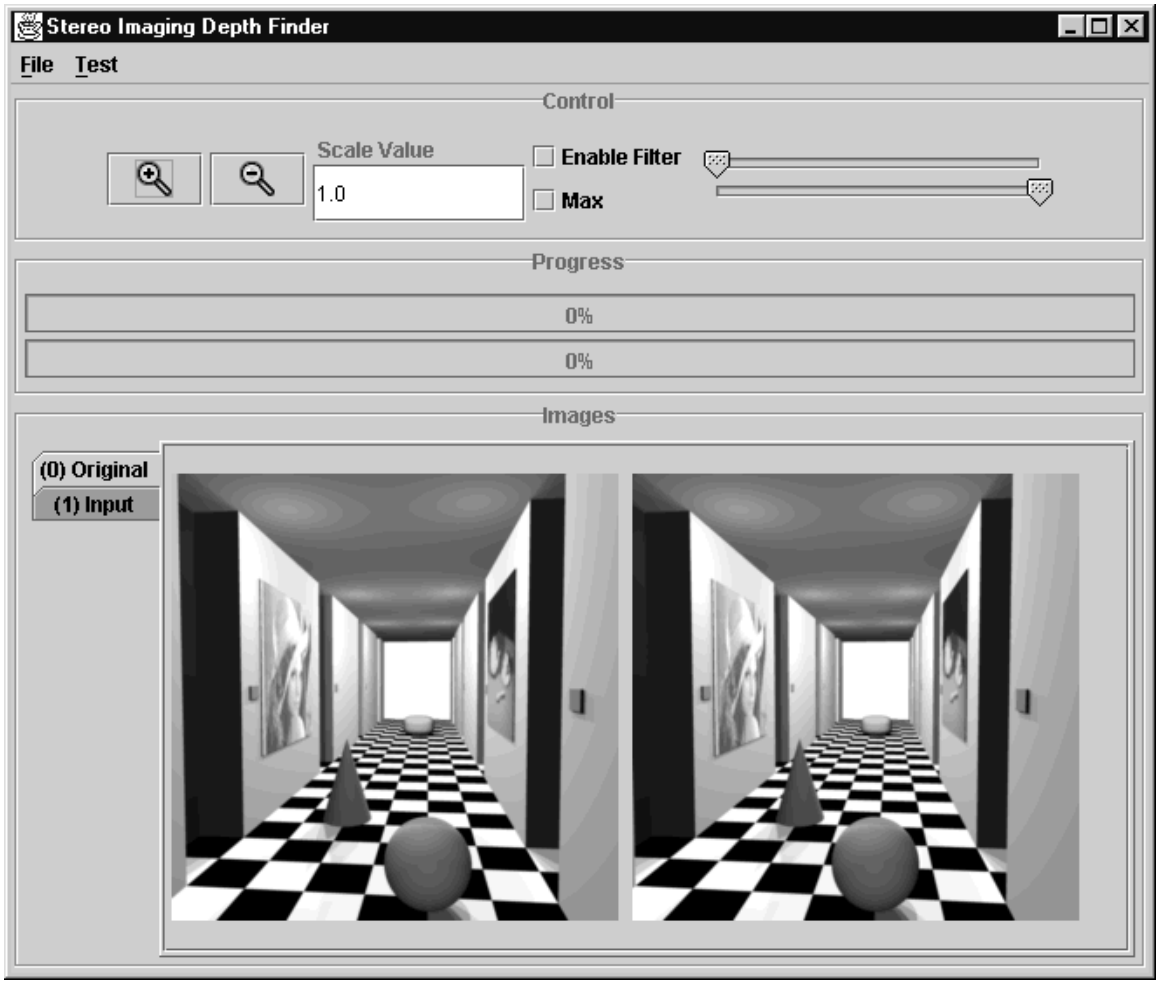

Figure 6-3: Graphical User Interface for Stereo Depth Finder.

6.4 Wavelet Compression

The wavelet compression ACS Benchmark [KumP98] was also implemented using the Janus tools. Figure 6-4 diagrams a single-layer wavelet decomposition. Figure 6-5 shows the juxtaposition of three decomposition layers to create the overall wavelet compression transform. The wavelet compression application was implemented by two graduate students who were not directly involved in the construction and development of the tools. The ease with which they developed their application, at least anecdotally, demonstrates that the principles of Janus application design are relatively easy to understand and apply.

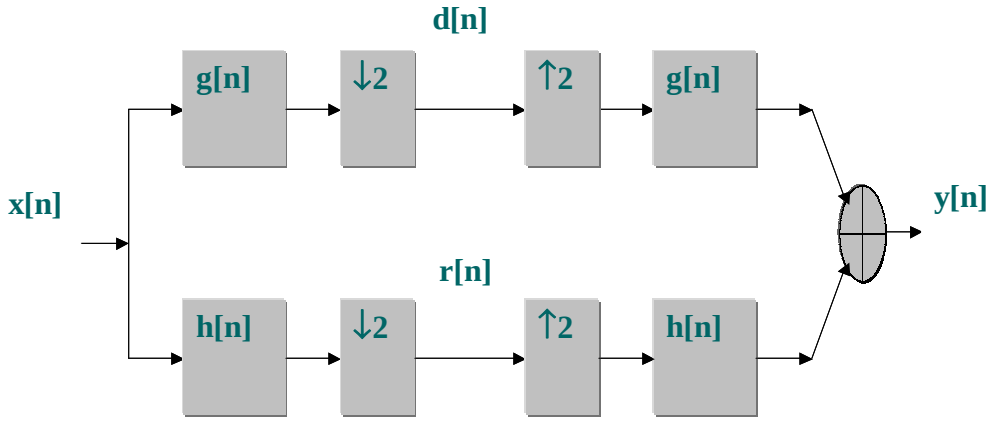

Figure 6-4: Wavelet Compression Decomposition Layer.

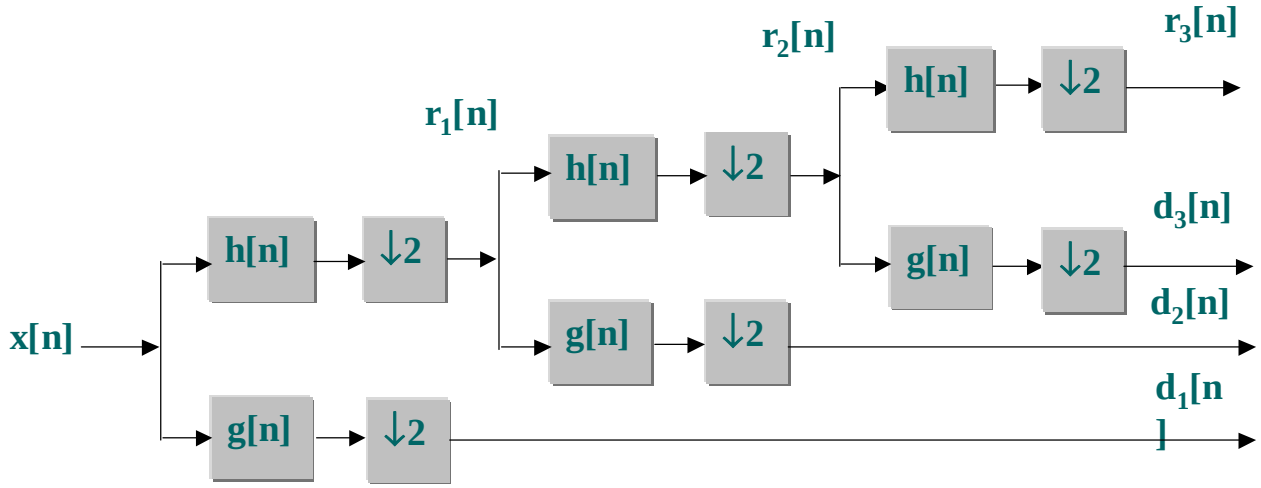

Figure 6-5: Three Layer Wavelet Compression Transform.

Chapter 7 Results

This final chapter reexamines the design goals of the Janus tools and assesses how well they have been addressed. The successes and limitations of the tools are presented. Where limitations are presented, design options for further exploration of the design space are suggested. Lastly, some general philosophical conclusions are offered about the directions that future versions of the Janus tools might pursue.

7.1 Goals Revisited

Section 2.4 presented the four design goals that shaped the development of the Janus tool set. Chapter 1 itemized the resulting contributions made by pursuing these goals. This section will review these goals again in the context of the following claimed contributions:

- A demonstration, using real applications of interest, of a unified design framework for developing RTR CCM applications
- An object-oriented approach that allows developers to build applications without knowledge of the architecture, compilation methodology or run-time implementation of the target CCM
- Examples of how designs can be automatically adapted to take advantage of the features available within a target architecture
- A stage-based design paradigm that provides an intuitive approach to designing RTR applications
- An evaluation of the successes and shortcomings of the approach

7.1.1 Unified Design

Janus' primary design goal is to allow developers to construct their applications using a single design specification. This design specification must allow developers to specify both the host-side and CCM-side behaviors of their application using a single design representation. The Janus tools have met this goal with complete success.

All of the applications presented in Chapter 1 were constructed using an integrated Java development environment. The user interfaces were constructed using all of the graphical facilities available in the Java toolkits. The ordering of computational stages were performed

using specialized Java classes. The hardware circuits were designed in JHDL. All of the Janus tools' underlying compile-time and run-time code was written in Java. The only portion of the entire system that is not written in Java is the interface to the target hardware's vendor libraries. This code, however, constitutes the hardware backend and is invisible to application developer.

There are many advantages to the integrated representation. The user is relieved of the burden of writing host-side control code that interfaces with the CCM vendor's interface libraries. Janus automates all of the configuration and data transfer operations. Bringing the hardware description into the domain of a fully featured language has many intriguing opportunities. Reusable hardware modules that would be a semantic nightmare to write in VHDL become approachable problems using the programming features of Java. Inheritance provides methods for reusing common functionality between designs. Abstract creation of objects would enable objects to examine their available resources and make time-space tradeoffs.

Janus also addressed the most important and pragmatic reason for the integrated representation. It allows the designer to represent designs with multiple configurations. The Janus tool's stage objects allow programmers to assemble hardware objects into a computational hierarchy. Further, it does not require the user to specify the exact execution order of the computations or the specific resources upon which they must execute. This order is dependent on the target hardware and the Janus tools automatically adapt the user's design appropriately.

7.1.2 Virtual Hardware

Both the image interpolator application and the stereo depth-finding application demonstrate the Janus tools' ability to create designs that utilize virtual hardware. Virtual hardware allows designers to address problems whose computational needs exceed the resources of the available CCM. Villasenor's automatic target recognition application is an example of an application that utilizes this capability. The virtual hardware model also allows application designers to build solutions that have procedural elements, but still exploit the parallelism of algorithms. The image interpolation application and the stereo depth finding application are examples of this.

Both the image interpolator and the stereo depth finder are multistage algorithms. The natural separation between stages is the retiring of an intermediate image or data item to memory. It is certainly possible to implement both applications using ASIC-style configure-once designs. The static design style, however, requires the user to partition the available resources between the various stages. For instance, an ASIC-style design for the image interpolator built on a WILDFORCE might utilize one processing element for each of the four stages.

Assume that the image interpolator is processing a single image. Also assume, for the moment, that the target architecture has negligible run-time reconfiguration overhead. Let the compute time for processing a set of IF rows and a set of IF column be *t*. Let the time required to compute a set of FST rows be *4t*. Finally, let the time required to compute a set of FST columns be *16*. Given the data dependencies of the computation, no subsequent stage of the computation can begin before the previous stage has completed. Assume that each processing element only has enough resources to implement one instance the stage assigned to it.

The ASIC-style implementation must bring the image onto the first processing element and complete its computation before passing the image on to the next processing element and so on throughout the computation. Thus the total compute time for the ASIC-style computation is the sum of the individual compute times, or *22t*.

The RTR-style implementation may utilize all of the CCM's processing resources for each stage of the computation. This allows the RTR design to implement four parallel row or column operations for each stage of the process. This parallelism decreases the compute time for each stage by four. So, the RTR-style implementation has a compute time of *5.5t*. The RTR-style design is *16.5t* faster than the ASIC-style design.

The ASIC-style design is so much slower because it lacks parallel elements. If, instead of a single image, the input to the image interpolator is a series of images, the ASIC-style implementation can operate its stage in pipeline fashion and improve its performance. The pipeline can produce a result as fast as the slowest computational unit can produce a result. So, even with pipelined parallelism, the ASIC-style design is limited by the final stage to a *16t* throughput that is still *10.5t* slower than the RTR-style design.

There are further optimizations that can be performed on the ASIC-style design. The computational resources can be reallocated. The row and column operations for the IF and FST, respectively, are identical except for the memory scan pattern that provides their input data. One processing element could be dedicated to perform IF row and column operations. Another could be used to do the FST row operations. That leaves two processing elements to perform the costly FST column operation. With alternating frames of data going to each of the two FST column units, the processing time of the ASIC-style design could be reduced to *8t*.

With even greater care, the ASIC-style design could probably be made competitive with the RTR design. However, which design is easier for the application developer to describe and which design is easier to automate? One of the conclusions of the Janus development effort is that the introduction of this procedural element to hardware design makes automation of designs easier. In addition, application developers probably begin their design effort with a procedural description of the application in either MatLab or C. The procedural nature of the RTR-design style makes it easier for the application designer to translate from the microprocessor development environment to the CCM domain.

7.1.3 Pragmatism

The design and development of the Janus tools were governed by the simple principle that the result of the research should be a functional tool that adds value to the design process. The design and implementation of the XC6200 based image interpolator presented in [HudL96] took months to complete. The implementation of the image interpolator using the Janus tools took approximately two weeks, including time spent debugging the Janus framework itself.

In some ways, this is an “apples and oranges” comparison. The arithmetic in the earlier interpolator is serial while the Janus implementation is parallel. The design tools for the XC6200 platform are not very well developed while the XC4000 series tools are very well developed. However, while some of the basic underlying technologies made the XC4000 series design easier, the Janus application was also in many ways far superior to the early implementation. For instance, the Janus version can handle different sized input images and has a graphical user interface that interacts with the hardware application.

While there are some restrictions on the way that operations can be formulated and on the manner in which they can be associated to build an application, three fully functional applications have been built using the Janus tools. This demonstrates that the tools are capable of producing complete designs. There also is anecdotal evidence from the design team that it is significantly easier to build applications using Janus than with methods that are more traditional.

The remaining question of pragmatism is one of application performance. The run-time behavior of Janus applications is characterized by the following equation.

$$t_{overall} = \sum t_{config} + t_{fetch} + t_{compute} + t_{retire} + t_{software}$$

The overall run time, $t_{overall}$, is the sum of the amount of time spent configuring processing elements (t_{config}), moving data to the accelerator card (t_{fetch}), computing results ($t_{compute}$), moving results back to the host (t_{retire}), and executing software running on the host ($t_{software}$).

The first element of the sum, t_{config}, is governed by the design of the underlying FPGA, the design of the accelerator card, and some characteristics of the application itself. The following equation expresses the relationship between these parameters.

$$t_{config} = C \bullet N \bullet t_{program}$$

The $t_{program}$ term represents the amount of time required to program a processing element. A XC4062 requires approximately one and a half million bits of configuration data. The chip is programmed serially with a maximum rate of approximately 15 MHz. Therefore, reconfiguration of a XC4062 takes at least 100 ms assuming no overhead for moving the configuration from the host to the CCM. Empirical testing on the WILDFORCE architecture places $t_{program}$ on the order of 200 ms.

The N term represents the number of PEs being used in an application. The WILDFORCE architecture is capable of programming one PE at a time. Therefore, if four PEs are being used, they must be programmed sequentially and the time penalty for doing so is multiplied by four. The C term represents the number of stages in an application. All of the utilized PEs must be reprogrammed for each stage of computation. A four-stage application like the image interpolator, multiplies the configuration time for the utilized PEs by another factor of four.

Table 7-1: Image Interpolation Run Times Varying Input Size 1 PE.

Input Image Size	t_{config} (s)	t_{fetch} (s)	$t_{compute}$ (s)	t_{retire} (s)	$t_{software}$ (s)
64x64	0.7811	0.0802	0.0941	0.1021	0.0481
128x128	0.788	0.2115	0.2421	0.3467	0.019
256x256	0.7787	0.803	0.4578	1.2744	0.048
512x512	0.7764	3.0679	1.0676	4.8881	0.1103

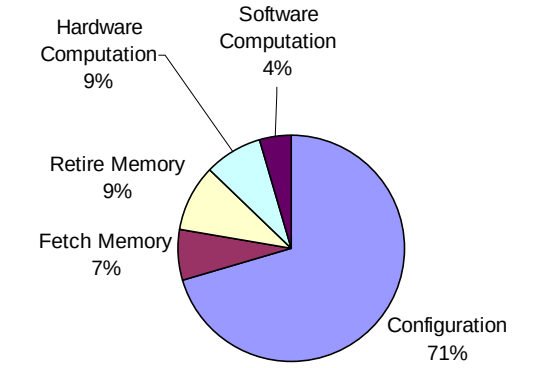

Figure 7-1: Interpolation 64x64 1 PE.

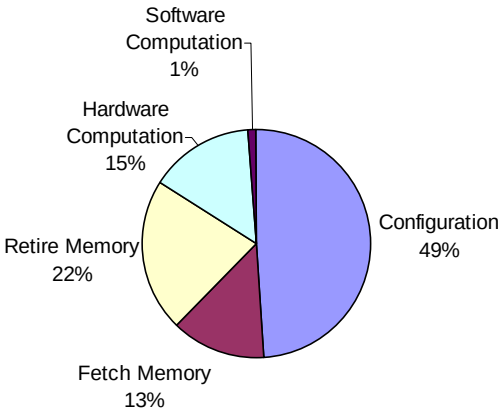

Figure 7-2: Interpolation 128x128 1 PE.

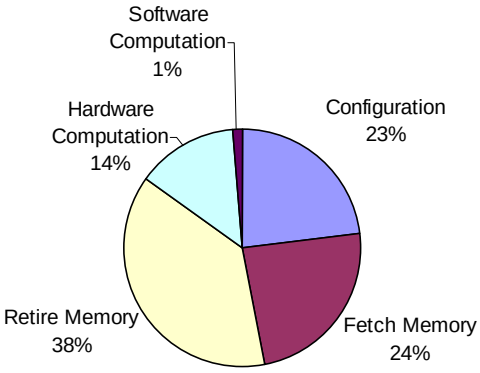

Figure 7-3: Interpolation 256x256 1 PE.

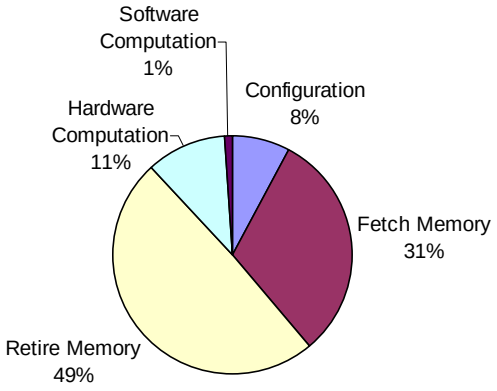

Figure 7-4: Interpolation 256x256 1 PE.

The relationship between t_{config} and the other times in the system is the critical factor for judging the performance of a Janus application. There is a preponderance of evidence in the literature that CCM accelerators are capable of outperforming microprocessors for a large class of computations. Accordingly, the focus of this research is not the optimization of memory transfer between host and accelerator or of the computational datapath itself. The focus is on examining the impact that the design methodology's inherently reconfigurable nature has on performance. The factor present in Janus designs that is not present in static designs is reconfiguration overhead. The ratio of t_{config} to $t_{overall}$ is the configuration overhead and is the metric of interest. It

represents the amount of time that the application spends performing tasks that are not computationally useful.

Table 7-1 provides some performance data from the image interpolation application running an an Annapolis Micro Systems WILDFORCE board. The design runs at 25 MHz. The results are averages from a series of runs. The data is for the image interpolator processing four different sized images using a single PE. Figures 7-1 through 7-4 illustrate relationships between the various run times for each of the different image sizes.

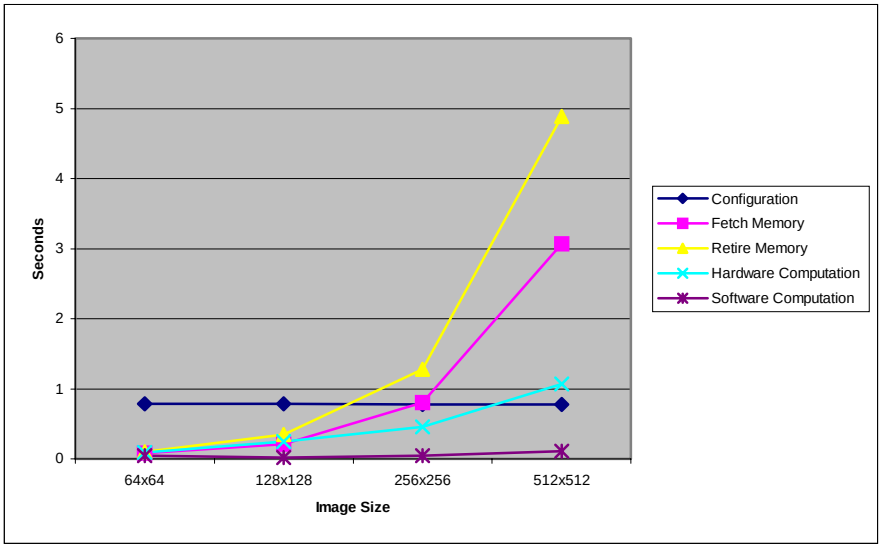

Figure 7-5: Run Times with Varying Input Sizes.

Figure 7-5 graphs the various components of the overall run time. As the size of the input image increases, the amount of time spent moving data between the host and accelerator memory and the amount of time spent performing computation both rise. However, the amount of configuration time remains constant. So, while the image interpolator's configuration overhead for the 64x64 input case is around 70%, the configuration overhead for the 512x512 input case is only 8%. Figure 7-6 shows this trend.

Further decreases in configuration overhead can be achieved by increasing the amount of computation performed during each stage of the computation. If, for example, the image interpolation application was being used to interpolate a video stream and the application could tolerate the latency of buffering up several frames of data, even lower configuration overheads can be achieved. Instead of processing a single frame of data during each stage of computation, the application could process several. The resulting configuration overhead curve would look like Figure 7-7. As the number of buffered frames increases, the original configuration overhead of 25% drops to below 5% for the eight-frame case.

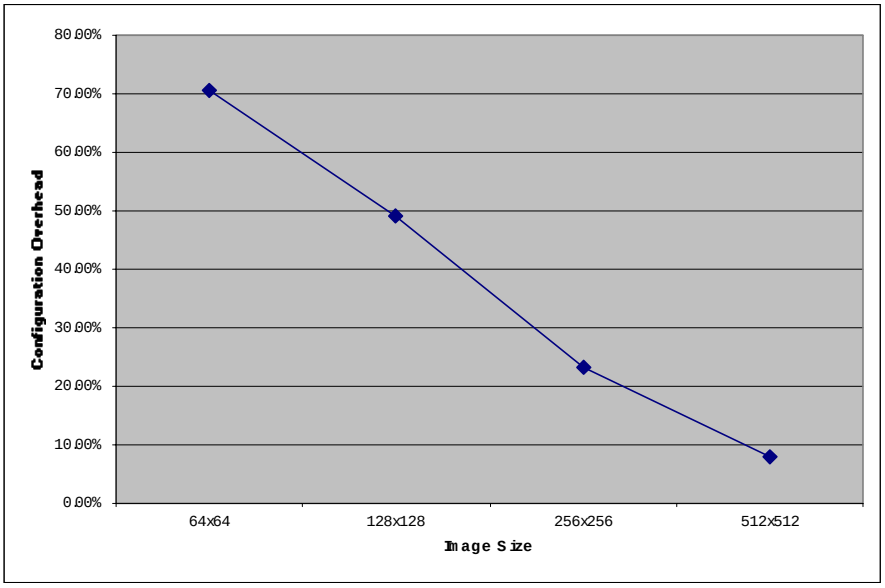

Figure 7-6: Configuration Overhead with Varying Input Sizes.

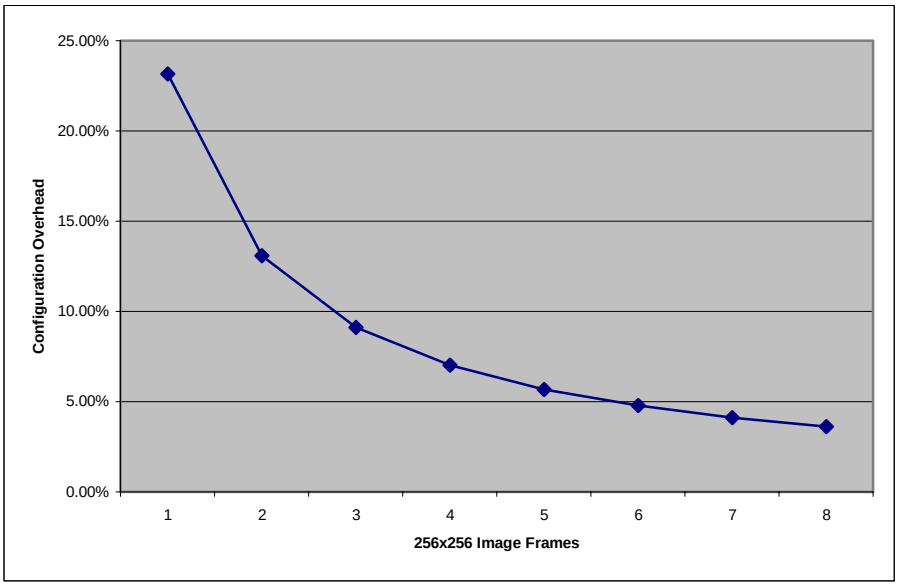

Figure 7-7: Extrapolated Configuration Overhead for Multiple Frames.

Another pragmatic issue of Janus application performance is the tool's ability to increase the performance of applications by exploiting the available parallel resources on a platform. Figure 7-8 shows the run time results for N = 1, 2, 3 and 4. There is a dramatic increase in configuration run time as predicted by the t_{config} equation. Each additional PE used by the system requires another $C \bullet t_{program}$ seconds to configure the PEs. This particular weakness in application

performance is a weakness in the underlying architecture. This effect could be eliminated with some simple architectural changes to the WILDFORCE board. Parallel programming of the processing elements, especially in the case of a Janus application where the same configuration is used for all four elements, is technically feasible and would eliminate this effect.

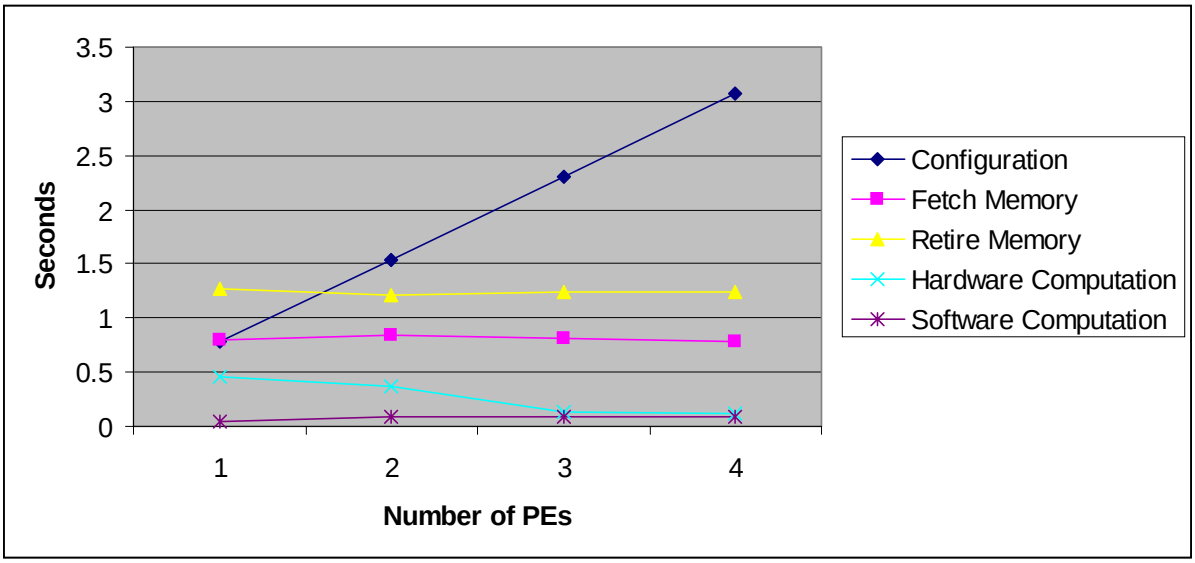

Figure 7-8: Run-Time Characteristics with Multiple PEs.

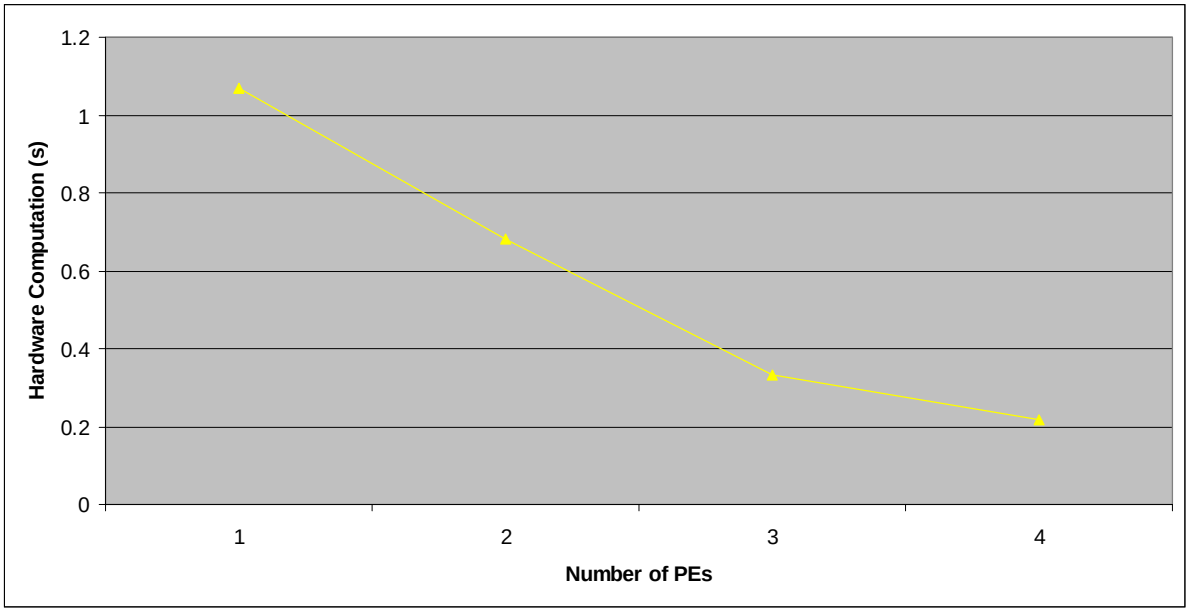

Figure 7-9: Hardware Computation Time with Multiple PEs.

Given the mitigation of configuration overhead by allowing parallel programming of the processing elements, Figure 7-9 shows the linear decrease in computation time gained by using multiple PEs. Figure 7-8 shows that memory access times remain constant, which is expected since adding parallel computation elements does not change the amount of data that must be moved back and forth between the host and the accelerator.

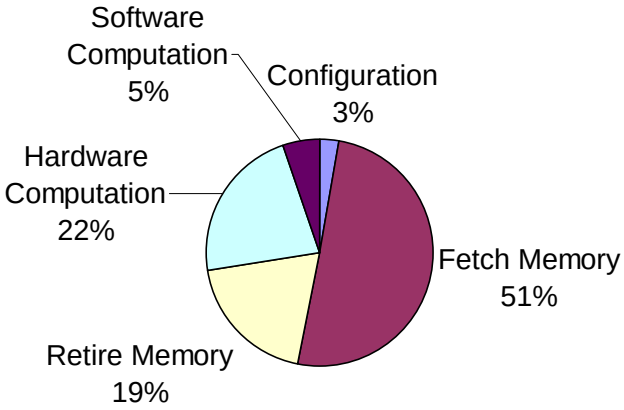

Figure 7-10: Stereo Depth Finding 1 PE.

Figure 7-10 shows the run-time distribution for the stereo depth finding application. The stereo depth finding application performs many two-dimensional cross-correlation operations on the stereo pairs. This processing is more intensive than that required in image interpolation, so the application has a much lower configuration overhead even for relatively small images. The stereo imaging application also performs some of its stages in software on the host. This accounts for the larger size of the software slice. The cross-correlation operation requires an image as input and produces a set of coordinates as outputs. This results in the large difference between the fetch and retire slices.

Figure 7-11 illustrates the run-time behavior of the wavelet compression application. The wavelet compression application demonstrates a situation where not only the configuration overhead is small with respect to the computation, all other facets of run time are small with respect to the computation.

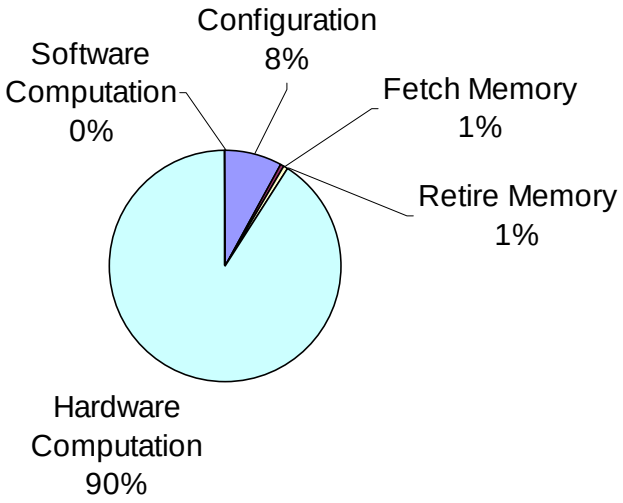

Figure 7-11: Wavelet Compression in 1 PE.

7.1.3.1 Further Strategies for Reducing Run-Time Overhead

Several different strategies for handling the run-time overhead of reconfiguring large FPGAs have emerged in the literature. One very promising strategy is the multiple-context FPGA. Originated by [OngR95], the idea of a context switching FPGA is discussed in [TriC97] and the implementation of such a device is reported in [ScaV98]. The principle of the devices is relatively simple. FPGAs store their configurations in onboard SRAMs. The silicon area required to implement the SRAM configuration bits for a configurable unit is smaller than the amount of silicon required to implement the unit itself. Including additional configuration memory on an FPGA allows the device to support multiple versions of a single configurable unit. In the device reported by [ScaV98], four separate configurations can be maintained on chip simultaneously. The one that is currently controlling the configurable units is the active configuration. The device described in [ScaV98] is capable of changing its configuration, e.g. its context, on a single clock edge. In addition, the device from [ScaV98] is capable of loading new configurations into an SRAM while another configuration is actively computing. This allows “background” loading of configurations while other processing is occurring.

The reconfiguration overhead problem is diagramed in a timeline form as Case A in Figure 7-12. In Case A, compute time is largely overshadowed by reconfiguration and the application is not effectively utilizing its resources.

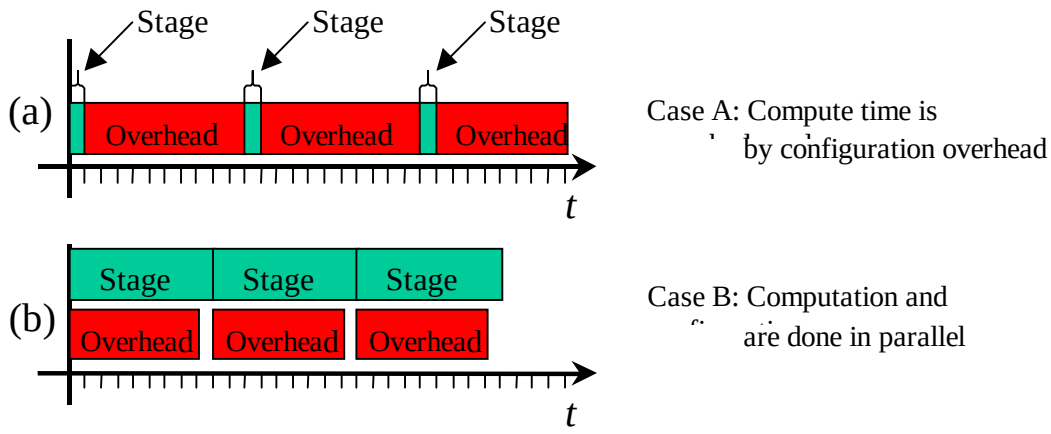

Figure 7-12: Configuration Overhead with Respect to Computational Time.

For simple applications, like the four-stage image interpolator, multi-context devices provide a perfect solution. If a multi-context device supports four on-board configurations, then all four stages of the image interpolator can be present on the device at the same time and the overhead for changing configurations is on the order of a clock period. When the number of stages in a application exceeds the number of configurations available on the multi-context device, new configurations need to be loaded onto the device during execution. The ability to load new configurations onto the device in the background, while computation is going on in the foreground, allows applications to adopt configuration cycles like the one shown Case B in Figure 7-12. The amount of time required to load a configuration on a multi-context device is still large, but since that operation can be overlapped with useful computation, the ratio between computational time and configuration overhead required for efficient operation is only 1:1 instead of the 10:1 that would be required in Case A.

7.1.4 Hardware Independence

Two concrete implementations of the Janus ***HardwareArchitecture*** class have been implemented as part of this research. One allows Janus applications to target an Annapolis Micro Systems WILDFORCE populated with XC4062XL FPGAs. The other targets Virginia Tech's Tower of Power (TOP) computing engine [JonS99]. The tower of power computing engine is a collection of sixteen Dell workstations networked together with Myrinet. Each workstation has a WILDFORCE board. Communication between the host application and the various TOP processing elements is performed over the network. Data must not only be moved from host to CCM, but from the central control point, to another computer and from that computer to the CCM. All of this extra control is transparent to the Janus developer. The difference between targeting the WILDFORCE and the TOP is a single menu selection during the compilation process.

In addition, the Janus tools automatically adapt applications to target architectures. The image interpolator, when targeted for the WILDFORCE runs four operations in parallel, while the TOP implementation takes advantage of up to sixteen processing elements running in parallel. The

adaptation of the application to the target architecture happens automatically when the developer compiles an application to a new target.

7.2 Limitations and Opportunities

While the first version of the Janus tools has met its goals of providing an easy to use, hardware independent development model for RTR applications, some compromises were made along the way. None of the compromises is without solutions but, in general, the solutions adopted in the first version of Janus are those that could be implemented with greatest ease and reliability. This section discusses what those compromises were and how future versions of the Janus tools might provide solutions that are more powerful.

7.2.1 Operator Memory Width

At the core of Janus' hardware independence is the specification of the operator memory interface. This interface must make some assumptions about the nature of memories in CCMs. Some of the assumptions are quite reasonable. For instance, the memory interface assumes that signals leaving an FPGA are registered at the chip boundary. Registering signals at the chip boundary introduces a two clock-cycle delay during a memory read. This is a standard design practice for high-performance FPGA designs and can be easily implemented on any CCM. Some of the assumptions are less innocuous. Janus application designers must make some assumption about the sizes of the memories that are available. The current implementation handles this limitation by allowing the application developer to require a minimum memory size for a target CCM. Applications may request information about the size of the target architecture from the hardware backend and produce an appropriate exception if the target system does not have sufficient memory. It is also possible to imagine, given the high-level interaction available within the Janus object model, operations that are capable of automatically sizing themselves based on the resources reported by the target hardware.

The most difficult assumption to make relates to the width of the memory's data path. Following the same policy that was established for memory size, the current Janus tools allow application developers to specify what memory data widths their application will support. The width of the data path can be queried from the backend. An attempt to compile an application on an architecture that does not provide a supported data width results in an exception. This places the burden of providing architecture independence on the application designer. There are alternative approaches.

The Janus tools could specify the width of a data port and have all applications, regardless of target architecture, target that single width. The tools would then be responsible for providing hardware adapters that would make the existing data ports behave as if they were of the specified width. Narrower memory interfaces would be time-multiplexed. Wider memory interfaces would either simply be truncated, which would waste a significant number of memory bits, or implement read-modify-write cycles to provide memory operations on appropriately sized portions of the larger words. Placing the responsibility for solving the memory width problem on the tools themselves provides greater automation, but runs the risk of creating severe performance problems that a simple change in the user's design might completely bypass.

Which solution is most appropriate has not yet been determined. Part of the decision may be made by the evolution of the CCM's themselves. CCM architectures may converge on a single appropriate memory width, in which case, allowing the user to create designs that are limited to a single memory interface is an acceptable solution. If CCM's do not converge on a single memory width, further research may favor the development of adaptable operators capable of interacting with the hardware environment and designing themselves appropriately.

7.2.2 Operator Size

The limited amount of reconfigurable resources available on the individual FPGAs that make up the CCM is another area where the real world intrudes upon theoretical architecture independence. Ideally, an application designer would be able to build operators without considering their resource consumption. The current version of the Janus tools allows designers to require potential computational platforms to have a minimum amount of available logic. This amount is a compromise between the prevailing size of FPGAs generally available on potential architectures and the convenience of partitioning the algorithm.

Alleviating this limitation leads to at least two very difficult problems. As the size of a theoretical operator grows it will first exceed the size of a single FPGA and then exceed the size of all the available logic on the CCM. When the size of an operation exceeds the capacity of a single FPGA, spatial partitioning of the operator is required. This involves dividing the vertices of the operator's netlist into *n* sets, where *n* is less than or equal to the number of FPGAs in the system. The difficulty arises when some constraints are placed on the formation of the sets. Each vertex of the netlist corresponds to some logic. The sum of the logic in each set must be less than or equal to the amount of logic available on the corresponding FPGA. In addition, FPGAs in CCMs typically have limited interconnection resources between them, so the edges that connect vertices in different sets, which correspond to signals that must leave one FPGA and enter another, must correspond to wires that are actually available in the system. Solutions to this problem are an active area of research.

When the size of the operator expands beyond the capacity of all of the FPGAs available on the CCM, temporal partitioning of the design is required. Where a spatial partitioning of the problem took a large design and divided it up between two or more of the available FPGAs, temporal partitioning uses RTR to divide the problem up between subsequent stages. At this point, the problem appears to become recursive. The whole purpose of the Janus tools is to provide the user with a means of describing their multi-stage computation. Initially, it might seem that a good solution to the temporal partitioning problem would also solve the overall problem and therefore make the rest of the tools obsolete. It was this assumption that drove the aborted research described in [HudL98]. This precursor to the Janus tools attempted to solve the temporal partitioning problem at the application level. Instead of requiring that the user determine how to break the application broken down into stages, the way the Janus tools do, this initial tool tried to automate the decomposition. The research eventually revealed that the applications under study did not lend themselves to automated partitioning. The basic structure of the applications involved reading data from memory in a particular order, performing some relatively simple

computations on it, and retiring the results to memory in a sometimes substantially different order. The data reordering performed implicitly by fetching and retiring data to and from memory formed natural temporal boundaries within the computation. The “relatively simple” computations between these boundaries simply did not warrant automated temporal partitioning and automated discovery of the natural boundaries was deemed unnecessary since the divisions where obvious to the human designer’s eye. It was these conclusions that drove the development of the current Janus tools.

Within the context of the Janus design environment, the problem of a single operator exceeding the computational bounds of a temporal stage would correspond to an operator that was not “relatively simple”. If the operation required a large data reordering within itself, then the user would be able to partition it using standard Janus methods. If the operation actually exceeds the computational capacity of a temporal stage without a natural temporal boundary, then the investigation of an automated means of temporally partitioning it becomes a viable area for exploration. This remains one of the opportunities for future research within the Janus tools.

7.2.3 Operator Types

Currently, users of the Janus tools are restricted to specifying a strict ordering of computational stages. The assembly of stages during the build stage of the application forces the user to specify a single possible ordering for the execution of computational stages. It is easy to envision some simple applications whose data dependencies would allow some stages to execute simultaneously. It would be an easy task to remove this restriction and allow the user to specify sets of stages that could be performed simultaneously. One simple way to implement this would be to allow **StageUnordered** objects to contain other **Stage** objects.

Removing the restriction on the user is simple. How the tools handle the resulting specification is another matter. The simplest approach would be to simply flatten any **Stage** objects found in a **StageUnordered** object into a **StageOrdered** object with an arbitrary ordering of the stages. The tool could then proceed as before.

An arbitrary ordering is not the only option however. It is quite conceivable that the most efficient way to perform the computation is to divide the computation resources of the platform between the different stages and evaluate some of the stages in parallel. This is a standard scheduling problem of operations to limited resources with a goal of minimizing compute time.

A similar restriction with a similar solution is the removal of the requirement that **StageUnordered** objects contain operations of only one type. Lifting this restriction produces a very similar scheduling problem to the one created by allowing parallel stages. Solutions to both problems represent possible refinements of the Janus tools.

7.2.4 Operator Ordering

In their current manifestation, the Janus tools generate a script describing the exact sequencing of run-time events at compile time. One consequence of this restriction is that the application cannot perform any kind of data-dependent branching or looping. This constraint is also reflected

in Janus' representation of ordered stages. There are no provisions in the application building model for applications that iterate over data or that follow alternate execution paths dependent upon results computed from input data.

The current run-time system creates this limitation. The restriction could be removed by implementing a run-time system that provides the ability to schedule the execution of operations at run time. There is nothing in the current implementation that precludes such a run-time system. Indeed, it would facilitate a more dynamic nature even for applications as simple as the image interpolator. Currently, a change in the size of the input image requires a recompile of the interpolator. With a run-time scheduler, it would be possible for the system to handle the adaptation of the application to varying image sizes without a recompile.

A run-time scheduler would also allow applications to opportunistically exploit data dependent variations in the execution times of operations. Currently, the run-time system starts some number of operations running, waits for them to finish, and then schedules the next set. This is acceptable for operations like the ones in the image interpolator and the stereo depth finder. Each of these operations has the same execution time irrespective of the data upon which it is operating. If new application were developed, with operations whose execution time varied depending on their input data, this run-time strategy could result in a considerable inefficiency in the utilization of the computational resources. If, for instance, some number of these operations were started simultaneously and one of them took significantly longer than the rest, several processing elements would be sitting idle waiting for the longest operation to finish. With a run-time scheduler, it would be possible to start operations opportunistically whenever a processing element became free. Depending on the characteristics of the data dependent operations, this could constitute a considerable speedup.

7.2.5 Dynamic Operations

Finally, the Janus tools, as discussed in Section 2.4 do not provide a means for creating operations at run-time. The problem of creating hardware designs in response to a user request or a dynamic environment is an active area of research. Significant progress must be made in the areas of rapid place-and-route technologies before such technologies could be incorporated into the Janus tools.

7.3 Future Directions

Several other students have already created extensions to the Janus research during the time it has taken to write this dissertation. Some of their results have already been incorporated into this text. In this section, each is acknowledged in its own right.

7.3.1 Tower of Power Backend

David Lehn has created a Janus backend targeting the Tower of Power's (TOP's) Adaptive Computing Systems application programming interface [JonS99]. This backend demonstrates many of the architecture independent features of the Janus tools. The TOP architecture uses multiple microprocessor-based hosts integrated together using Ethernet and Myranet, so it is a

solid demonstration that Janus' architectural abstraction can be used to target very different systems.

7.3.2 Wavelet Compression Application

Zhimei Ding and Yifan Cheng collaborated on a wavelet compression application. The wavelet compression application is based on a four-stage process described in Cohen [CohD92]. Input images are processed through four Janus stages. First, the image undergoes a forward wavelet transform. The information in the transformed image is reduced by the adaptive quantization stage. The quantized image goes through a run-length encoding stage that provides compression. Finally, further compression is achieved during the entropy encoding stage.

7.3.3 Simulation of Multiple Memory Port Backends

During the development of the Janus tools, the only configurable platforms available in the Virginia Tech Configurable Computing Lab were based on the Annapolis Micro Systems WILDFORCE board, which supports only a single memory interface per FPGA. One interesting feature of the Janus development paradigm is the ability of the Janus compiler to adapt the implementation of applications to take advantage of architectural features in target platforms. One such architectural feature is the existence of multiple memory ports per FPGA. Since no such architecture was available for testing, James Atwell created a Janus backend that, using time multiplexing, simulated processing elements with multiple memory interfaces.

7.3.4 And Beyond…

The design and implementation of the Janus tools has attempted to examine the RTR application design process with a very pragmatic approach. The first and foremost goal was to create a useful design framework forming a basis for further automation research. The primary purpose of the Janus tools is to study the nature of run-time reconfigurable applications and determine what sort of automation is useful to the design flow.

The most exciting opportunity that the Janus tools provide is the means to explore capturing human design knowledge. Instead of taking the human out of the design flow, the Janus tools allow the designer to capture reusable design knowledge within their applications. This facility is provided, largely, by the use of a fully functional programming language as unified representation of the application. The unified representation allows the user to create hardware that interacts with portions of the design that were previously defined by a completely separate design flow. The fully functional programming language gives the user the ability to describe hardware modules that can adapt to their environment. Together these two abilities lay open an exciting opportunity for future exploration.

References

[BelH98] Peter Bellows and Brad Hutchings, "JHDL- An HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 175-184, April 1998.

[BooJ98] Grady Booch, Ivar Jacobson, and James Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley Publishing Company, 1998.

[BurD97] Jim Burns, Adam Donlin, Jonothan Hogg, Satnam Singh, and Mark de Wit, "A Dynamic Reconfiguration Run-Time System," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 66-75, April 1997.

[BreG97] Gordon Brebner, "The Swappable Logic Unit," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 77-86, April 1997.

[BYU99] Brigham Young University Configurable Computing Laboratory, *JHDL Documentation Tree*, unpublished technical report, April 1999.

[CasP98] D. Castanon, M. Petersen, S. Ponnuswamy, B. VanVoorst and C. Nanavati, "Constant False Alarm Rate Benchmark Specification," *Technical Information Report*, Honeywell Technology Center, March 1998.

[ChuW98] Michael Chu, Nicholas Weaver, Kolja Sulimma, Andre Dehon and John Wawrzynek, "Object Oriented Circuit Generators in Java," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 158-166, April 1998.

[CofL88] E. Coffman Jr., G. Lueker, and A. Rinnooy Kan, "Asymptotic Methods in the Probabilistic Analysis of Sequencing and Packing Heuristics," *Management Science.*, 34, pp. 266-290, 1988.

[CofS93] E. Coffman, Jr. and P. Shor, "Packings in Two Dimensions: Asymptotic Average-Case Analysis of Algorithms," *Algorithmica*, vol. 9, no. 3, pp. 253-277, 1993.

[CohD92] A. Cohen, I. Daubechies, J. Feauveau, "Biorthogonal Bases of Compactly Supported Wavelets," *Comm. Pure Appl. Math.*, vol. 45, 1992.

[ClaH96] David Clark and Brad Hutchings, "Supporting FPGA Microprocessors through Retargetable Software Tools," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 195-203, April 1996.

[DehA96] Andre DeHon, "DPGA Utilization and Application," *International Symposium on Field-Programmable Gate Arrays*, pp. 115-121, 1996.

[DunH98] Andrew Duncan, David Hendry, and Peter Gray, "An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 106-115, April 1998.

[FerP97] Leonard Ferrari and Jae Park, "An Efficient Spline Basis for Multi-Dimensional Applications: Image-Interpolation," *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 757-760, 1997.

[FowS97] Martin Fowler, Kendall Scott, Ivar Jacobson, *Uml Distilled : Applying the Standard Object Modeling Language*, Addison-Wesley Publishing Company, 1997.

[GamH95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, 1995.

[GarJ79] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[GehL96] Stephan Gehring and Stefan Ludwig, "The Trianus System and Its Application to Custom Computing," *6th International Workshop on Field-Programmable Logic and Applications*, Darmstadt, Germany, September 1996.

[GosJ96] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley Publishing Company, 1996.

[GorR98] Robert Gordon, *Essential JNI: Java Native Interface*, Prentice Hall Computer Books, 1996.

[GraD98] Douglas Grant, *Velab Release Notes*, Xilinx Inc., http://www.xilinx.com /apps/velabrel.htm, 1998.

[GunM96] Bernard Gunther, George Milne, and Lakshmi Narasimhan, "Assessing Document Relevance with Run-Time Reconfigurable Machines," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 10-17, April 1996.

[FelR94] James Feldman and Charles Retter, *Computer Architecture: A Designer's Text Based on a Generic RISC*, McGraw-Hill Computer Science Series, 1994.

[EdiH42] Edith Hamilton, *Mythology: Timeless Tales of Gods and Heroes*, Mentor, 1942.

[HauL98] Scott Hauck, Zhiyuan Li, and Eric Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 138-146, April 1998.

[HudL96] Rhett Hudson, David Lehn, and Peter Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 88-95, April 1996.

[HudL98] Rhett Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring, and Peter Athanas, "Spatio-Temporal Partitioning of Computational Structures onto Configurable Computing Machines," *SPIE Proceedings*, Vol. 3526, p. 62-71, October 1998.

[HutB99] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, and Mike Rytting, "A CAD Suite for High-Performance FPGA Design," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.

[IssW96] Tsuyoshi Isshiki and Wayne Wei-Ming Dai, "Bit-Serial Pipeline Synthesis for Multi-FPGA Systems with C++ Design Capture," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 38-47, April 1996.

[JacB98] Ivar Jacobson, Grady Booch, James Rumbaugh, *Unified Software Development Process*, Addison-Wesley Publishing Company, 1998.

[JonS99] Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott, "Implementing an API for Distributed Adaptive Computing Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.

[KumP98] S. Kumar, L. Pires, D. Pandalai, M Vojta, J. Golushky, S. Wadi, "Benchmarking Technology for Configurable Computing System," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[LemM95] Eric Lemoine and David Merceron, "Run Time Reconfiguration of FPGA for Scanning Genomic DataBases," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 90-98, April 1995.

[LukS96] Wayne Luk, Nabeel Shirazi, and Peter Cheung, "Modelling and Optimising Run-Time Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 167-175, April 1996.

[LukS97] Wayne Luk, Nabeel Shirazi, and Peter Cheung, "Compilation Tools for Run-Time Reconfigurable Designs," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 56-65, April 1997.

[MenM98] Oskar Mencer, Martin Morf, and Michael Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 167-174, April 1998.

[NatL99] S. Natarajan, B. Levine, C. Tan, D. Newport and D. Bouldin, "Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems," *Proceedings 1999 Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD)*, pp. 101-107, 1999.

[OngR95] R. Ong, *Programmable Logic Device Which Stores More Than One Configuration and Means for Switching Configurations*, U.S. Patent 5,426,378, 1995.

[PatC00] Cameron Patterson, "High Performance DES Encryption in Virtex FPGAs using JBits, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000.

[PetO96] James Peterson, R. Brendan O'Connor and Peter Athanas, "Scheduling and Partitioning of ANSI-C Programs onto Multi-FPGA CCM Architectures," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 178-179, April 1996.

[RumJ98] James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Publishing Company, 1998.

[SinH96] Satnam Singh, Jonathan Hogg, and Derek McAuley, "Expressing Dynamic Reconfiguration by Partial Evaluation," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 188-194, April 1996.

[ScaV98] S. Scalera, J. Vazquez, "The Design and Implementation of a Context Switching FPGA", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 78-85, April 1998.

[ShiL98] Nabeel Shirazi, Wayne Luk, and Peter Cheung, "Automating Production of Run-Time Reconfigurable Designs," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 147-156, April 1998.

[TauE95] E. Tau, I. Eslick, Chen, J Brown, and A. DeHon, "A First Generation DPGA Implementation," *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pp. 128-143, May 1995.

[TriC97] Steve Trimberger, Dean Carberry, Anders Johnson and Jennifer Wong, "A Time-Multiplexed FPGA," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 22-28, April 1997.

[VilS96] John Villasenor, Brian Schoner, Kang-Ngee Chia, Charles Zapata, Hea Joung Kim, Chris Jones, Shane Lansing, and Bill Mangione-Smith, "Configurable Computing Solutions for Automatic Target Recognition," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 70-79, April 1996.

[VijG90] Gopalakrishnan Vijayan, "Partitioning Logic on Graph Structures to Minimize Routing Cost," *IEEE Transactions on Computer-Aided Design*, pp. 1326-1334, vol. 9, no. 12, December 1990.

[WenB96] Alan Wenban and Geoffrey Brown, "A Software Development System For FPGA-Based Data Acquisition Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 28-37, April 1996.

[Xili95] Xilinx Incorporated, *XC6200 Field Programmable Gate Arrays*, 1997.

[Xili97] Xilinx Inc., *XC4000E and XC4000X Series Field Programmable Gate Arrays*, Nov. 10, 1997.

[Xili99] Xilinx Inc., *JBits*, http://www.xilinx.com/xilinxonline/jbits.htm, 1999.

Vita

Rhett Daniel Hudson was born on January 28, 1970 to Wayne and Bonnie Hudson. His early education was provided by Terraset, Haycock and Sunrise Valley Elementary Schools. He attended Langston Hughes Junior High and Eisenhower Middle School. In 1988, he graduated from South Lakes High School as an honored graduate.

From August 1988 through May 1999 Rhett studied at Virginia Tech. He has matriculated three times during that period. With his bachelor's degree in computer engineering in 1992, his master's degree in electrical engineering in 1994, and with his doctorate in electrical engineering in 2000.

His research focus in graduate school has varied from life-cycle complete product design, computer network support for remote design collaboration, intelligent computer agents, and digital libraries to his final choice: configurable computing. Working with Dr. Peter Athanas at the Virginia Tech Configurable Computing laboratories, Rhett developed the run-time reconfigurable application development environment known as Janus. This work formed the basis of his doctoral dissertation.

Rhett's professional career began as a Washington Post paper carrier, a job that he held for four years. During this period, Rhett also worked as a professional Cobol programmer for Vision Technologies where, after arguing with management about the shortsightedness of it, he wrote many a line of code that had no hope of functioning on January 1, 2000.

During his graduate career, Rhett worked as a graduate research assistant for Dr. Scott Midkiff on the SUCCEED Coalition's Deliverables Team-5, for which he served as Webmaster and general technical consultant. Later, he went to work for Dr. Athanas in the Configurable Computing lab.

Near the end of his graduate career, Rhett worked for the Advanced Engineering Design and Research (AEDAR) Corporation. He served as engineering manager for the corporation's research and development office.

At the time of this writing, Rhett has moved to Columbia, Maryland where he is employed by Annapolis Micro Systems as a principal design engineer in the applications group.