Structured Approach to Dynamic Computing Application Development

Stephen Douglas Craven

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

Doctor of Philosophy
in
Electrical Engineering

May 1, 2008
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, Reconfigurable Computing, Dynamic Computing, Development Environment

Structured Approach to Dynamic Computing Application Development

Stephen Douglas Craven

(ABSTRACT)

The ability of some configurable logic devices to modify their hardware during operation has long held great potential to increase performance and reduce device cost. However, despite many research projects and a decade of research, the dynamic reconfiguration of Field Programmable Gate Arrays (FPGAs) is still very much an art practiced by few. Previous attempts to automate the many low-level details that complicate Run-Time Reconfigurable (RTR) application development suffer severe limitations. This dissertation describes a comprehensive approach to dynamic hardware development, providing a designer with appropriate models for computation, communication, and reconfiguration integrated with a high-level design environment. In this way, many manual and time consuming tasks associated with partial reconfiguration are hidden, permitting a designer to focus instead on a design's behavior. This design and implementation environment has been validated on a variety of relevant applications, quantifying the effects of high-level design.

Acknowledgments

I owe a very large measure of gratitude to Dr. Peter Athanas. Without his efforts, I could not have entered Virginia Tech when I did. His guidance was essential in the development of this dissertation. Through his assistance I partook in a variety of experiences not normally granted graduate students, including teaching and proposal writing.

For his technical and career advice, I am indebted to Dr. Cameron Patterson. Our early morning conversations in the lab were a welcome source of inspiration.

I thank the rest of my committee, who each uniquely contributed to my progress. Dr. Patrick Schaumont provided insightful suggestions that significantly shaped certain aspects of this dissertation. Dr. Shawn Bohner graciously took time out of his busy schedule to provide advice on both research and career. I experienced Dr. Gary Brown's excellent service to the department first as a Bradley Fellow, which he oversees, and as a member of my committee. I am grateful for his support.

I owe much to everyone in the Configurable Computing Lab at Virginia Tech, many of whom I consider dear friends. Dr. Neil Steiner was a constant source of support and advice on all matters, technical and personal. Anthony Mahar was always a pleasure to work with. And Tingting Meng was willing to assist in any endeavor. Dr. Chris Anderson would always respond quickly to any radio-related emergency. His duties were gratefully replaced by Matt Shelburne upon his graduation.

Yousef Iskander deserves special recognition for the support and motivation he provided, without which this dissertation would have not occurred.

The staff at Impulse Accelerated Technologies, Inc., particularly David Pellerin and Brian Durwood, were especially helpful, providing access to the tools and code that made the implementation possible.

My present employer, Luna Innovations, Inc., and all of the engineerings in the Secure Computing Group must be thanked for tolerating my variable schedule and permitting me a leave of absence to complete much of the research.

My family's support, love, and prayers, though physically distant, were felt very strongly and greatly appreciated. The motivation and support provided by my dear friends, especially Kristina Karnes and the men of the Graduate Christian Fellowship, were instrumental in keeping me focused.

Finally, I must thank God for the amazing opportunities with which he has blessed me.

Contents

Acronyms

ASIC	Application Specific Integrated Circuit
CCM	Configurable Computing Machine
CSP	Communicating Sequential Processes
DSP	Digital Signal Processing
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLL	High Level Language
HLS	High Level Synthesis
HPC	High Performance Computing
JIT	Just-In-Time
KPN	Kahn Process Network
LUT	Look Up Table
RCSF	Reconfigurable Computing Specification Format
RTL	Register Transfer Level
RTR	Run Time Reconfiguration
SDR	Software Defined Radio

List of Figures

List of Tables

Chapter 1

Introduction

Configurable computing devices, such as Field Programmable Gates Arrays (FPGAs), permit arbitrary reprogramming of the devices' hardware functionality after manufacture. This programmability greatly decreases development costs and time-to-market compared to Application-Specific Integrated Circuits (ASICs). For many applications the drawbacks to FPGAs, lower performance and increased per device cost, are far outweighed by the fast implementation time — minutes to hours, instead of the months to years required to design and manufacture an ASIC.

Almost all commercial designs using FPGAs do not reconfigure the hardware once the product is deployed; however, previous research has demonstrated performance benefits from partially or fully reconfiguring an FPGA during operation [2] [3] [4]. Two mechanisms can be used to achieve a benefit from this Run-Time Reconfiguration (RTR): fine-grained circuit customization and virtual hardware. Dynamically creating custom digital circuits tailored to the task at hand; for example, by customizing an encryption circuit for a specific key, can provide significant performance improvements. Virtual hardware, on the other hand, permits a design to be larger than the available resources, with idle logic swapped out for active circuits. By removing configurable logic resource limitations, RTR can permit smaller and cheaper devices to be fielded.

Unfortunately, developing RTR applications has been a difficult undertaking. The reported successful RTR designs generally involved much low-level work by designers with detailed knowledge about an FPGA's inner workings. In spite of several attempts to abstract these lower-level details [5] [6] [7], no tools currently exist for modern architectures to assist a designer in these complex tasks. FPGA design tools share a lineage with ASIC tools, with both assuming static designs. What little vendor support that has been available forced the designer to do much manual, low-level work. Commercial tools in the form of design capture environments and simulators are non-existent. An FPGA designer developing an RTR application is very much a trailblazer, forced to improvise the required tools with little assurance that the final design will even function.

Seeing the need for an RTR application development methodology, several researchers over the past decade have proposed design environments and implementation flows supporting RTR. These projects range from paper proposals to integrated environments supporting both hardware and software development. More ambitious efforts perform automatic spatial and temporal partitioning. Acceptance of these methodologies, however, has been hindered for a variety of reasons; including esoteric design capture methodologies, architecture-dependence, and inferior tool performance.

Recent trends and technology advances have rekindled interest in RTR. The configuration architecture of the latest FPGAs is much more amenable to fast partial reconfiguration and, for the first time, true two-dimensional reconfiguration is available in high-performance mainstream devices. The inclusion of embedded processors has freed dynamic hardware designs from their previous reliance on a PC for configuration management, opening new application domains. The growing importance of FPGAs in Software Defined Radios (SDRs) [8] is prompting FPGA vendors to provide backend tool support for RTR [9].

The objective of this dissertation is to define an architecture-agnostic, structured approach to RTR application development, incorporating the latest advances and trends in configurable computing, to permit the rapid creation of dynamic hardware for streaming applications from a high-level specification. Previous research attempts, unaccepted even in their own time, were unable to capitalize on these current advances. Leveraging existing research, this approach enables the use of commercial design entry and simulation tools, fully incorporates embedded processors into the computational and programming models, and permits dynamic partial self-reconfiguration through an embedded configuration controller.

This research offers the following contributions:

- Creation of a design and implementation flow for streaming RTR applications. The project provides a unified development environment for both hardware and software through the seamless integration of embedded processors.

- Development of a practical RTR design and research environment. While the design flow can utilize a variety of tools for design capture, a commercial high-level synthesis tool has been extended to support the simulation and synthesis of RTR designs. This approach is in-line with previous research, where several projects attempted some form of high-level design entry. To avoid the development issues, performance limitations, and lack of acceptance previous projects encountered using custom high-level specification methodologies, a partnership with a commercial high-level synthesis tool company has been formed.

- Inclusion of partial reconfiguration into design abstractions. Few previous projects consider partial reconfiguration and those that do attempt to shield the developer from reconfiguration decisions. In contrast, this work provides high-level abstractions, permitting the developer to utilize partial reconfiguration to modify computation and communication structures at run-time.

- Demonstration of the benefits of RTR on modern FPGA architectures. To extend existing research detailing the performance enhancements of RTR to include the latest FPGA architectures, the design flow and environment have been used to implement encryption and image processing applications, domains well suited to the approach's computational model. These applications serve as benchmarks; with the design time and performance compared to traditional static designs as well as designs implemented using traditional RTR design flows.

Chapter 2 provides an overview of prior related work, including successful RTR designs and development environments for RTR applications. Attention is paid to the limitations of these flows along with methodologies for configuration management. Chapter 3 begins with an overview of the approach before expounding upon each stage. Chapter 4 details the realization of the approach, describing the design capture and implementation tools. Example applications created using this approach are discussed in Chapter 5 with performance and productivity results presented in Chapter 6. Finally, Chapter 7 discusses the project's conclusions and future work.

Chapter 2

Background

2.1 Reconfigurable Computing

Reconfigurable Computing (RC), also known as configurable computing or adaptive computing, is an area of computer engineering concerning computational devices that can be configured at the hardware level after manufacture [10]. General purpose processors, such as are found in desktop computers, can programmed via software to perform any computational task. However, the processor's hardware remains static – the number of multipliers and adders and the size of the cache can never be modified after the chip leaves the factory. Likewise, ASICs have a fixed hardware structure optimized for an application.

Several families of devices, however, may modify their internal hardware after manufacture. The most widely used RC devices are programmable logic devices, such as FPGAs. Other devices exist that provide more coarse-grained configurability. While many of these coarse-grained architectures have been proposed and implemented [11] [12], none have attained commercial success.

Initially used as glue logic for interfacing other digital components, FPGAs have evolved into stand-alone computational devices with performance that can exceed that of proces-

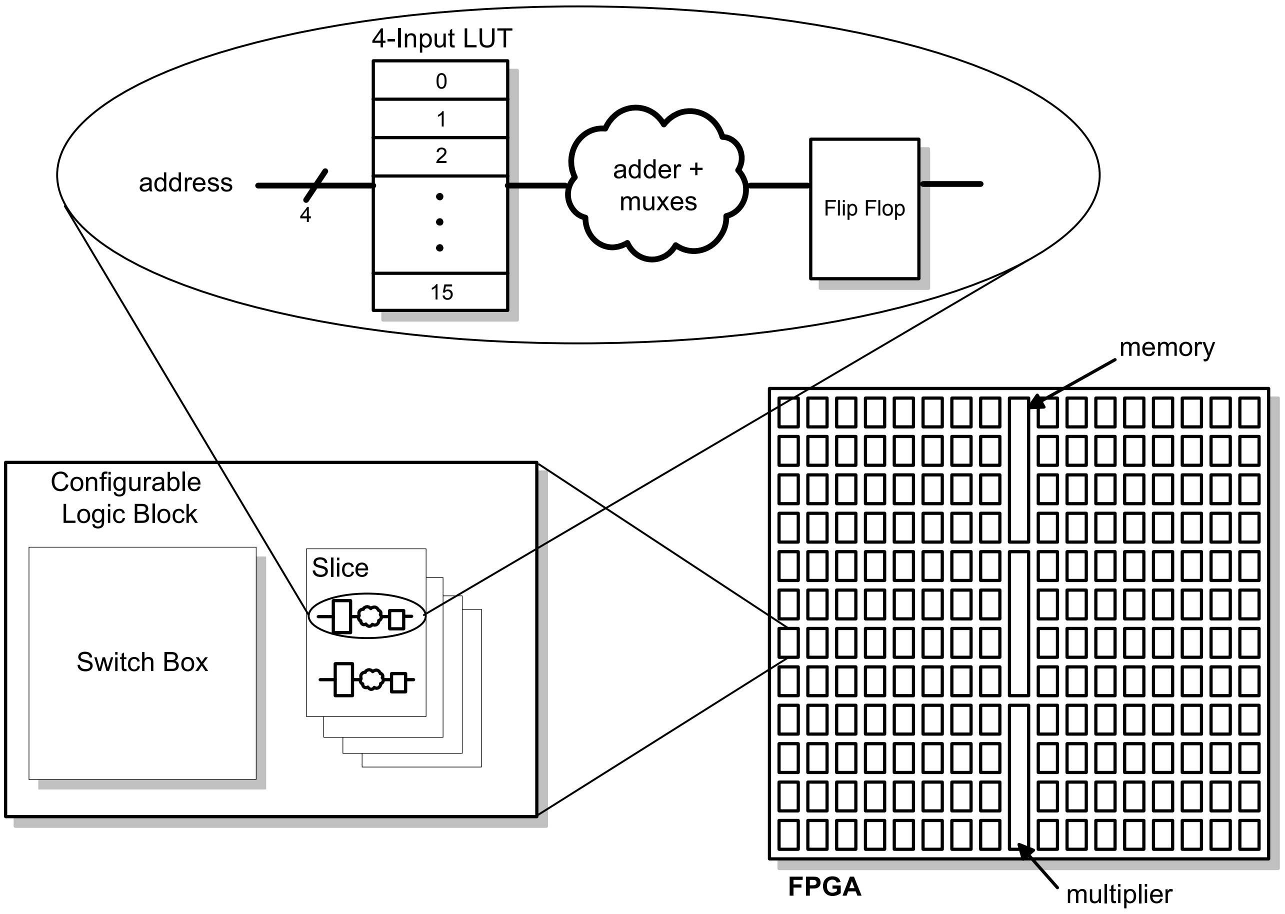


Figure 2.1: Typical FPGA structure.

sors in many applications. Figure 2.1 shows a typical FPGA's internal structure. A programmable Look-Up Table (LUT) acts as a logic element implementing any logic function of up to, in the case of the presented architecture, four variables or behaving as a 16-bit memory. In the Xilinx Virtex-II and Virtex-4 architectures represented in the figure, the LUT is paired with a flip flop storage element. Two of these LUTs are grouped into a slice. Four of these slices, along with a programmable switch box, comprise a configurable logic block. The configurability of an FPGA comes primarily from two elements: the LUTs configure the computational elements and the switch boxes configure the communication between LUTs.

Note that modern FPGAs include more coarse-grained computational and storage elements, such as dedicated multipliers and memories, called block multipliers and block RAMs, respectively, in the terminology of the largest FPGA vendor, Xilinx. Certain FPGAs, such as the Xilinx Virtex-II Pros and Virtex-4 FXs, also incorporate embedded processors in the sea of configurable logic. Additional processors, known as soft processors, may be created out of the configurable logic.

The fine-grained configurability offered by these architectures enable the hardware to be tailored specifically for the problem at hand. In applications that may be heavily parallelized or pipelined, FPGAs may greatly exceed the performance of processors in spite of their generally much lower clock frequencies. Domains that play to this strength of FPGAs are Digital Signal Processing (DSP) [13], cryptography [14], network applications [3], bio-informatics [15], and image processing [16].

The configuration architecture of Xilinx FPGAs is shown in Figure 2.2. The device is configured by the loading of configuration data in segments called frames. A frame runs vertically the entire height of the device for older Xilinx FPGAs. The newer Virtex-4 devices consist of multiple, independent frames per column. To program the device these frames must be loaded into the FPGA from the bitstream through one of several interfaces. One or two Internal Configuration Access Ports (ICAPs) exist inside the device to permit the device to control its own configuration. Using the ICAP, the FPGA may load in new modules stored as partial bitstreams in an external storage medium such as memory. Additional external configuration interfaces permit another device, such as a processor, to manage the configuration.

While for a specific hardware configuration an ASIC will always outperform an FPGA by a significant margin, design and initial manufacturing costs for modern, deep-submicron ASICs run into the millions of dollars, making ASIC cost prohibitive for all but high volume products. For low volume designs or for short time-to-market products where the long design, manufacturing, and testing delays of ASICs may be prohibitive, FPGAs become attractive.

The reconfigurability of FPGAs provides an additional advantage over ASICs. A design mistake that spells disaster for an ASIC [17] could be fixed in an FPGA with a corrected

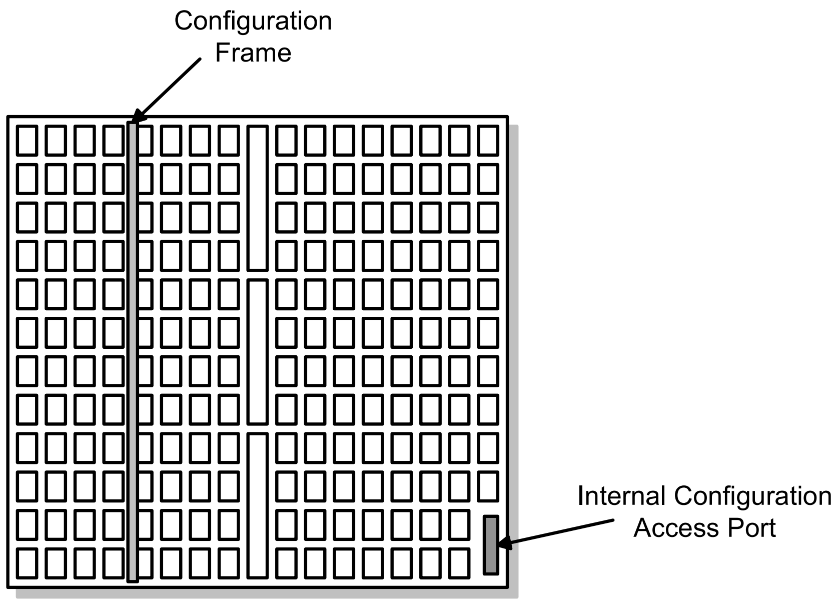


Figure 2.2: Xilinx Virtex configuration architecture.

configuration file. More interesting uses exist for this reconfigurability, however. By reconfiguring the device during operation the digital circuits can be tailored to the specific environment in which the FPGA is operating.

Many aspects of a hardware application can be reconfigured, from modifying a single gate to swapping out an entire soft processor. A taxonomy created by Verbauwhede and Schaumont [1] serves to illustrate and categorize the possible applications of RTR in hardware designs. Shown in Figure 2.3, this taxonomy maps the design space of RTR applications using three axes: architectural features, abstraction level, and configuration binding time. For a given architectural feature reconfiguration may be used at different levels of abstraction. Early RTR projects worked at the lower layers of LUTs and switches, with subsequent work raising the level of abstraction. The configuration binding rate corresponds to the rate at which configuration information is bound, or attached, to hardware. Static designs have a binding rate of zero – the configuration is fixed at design time. The binding rate is limited by the configuration speed of the FPGA and, in some designs, by the speed of the implementation tools.

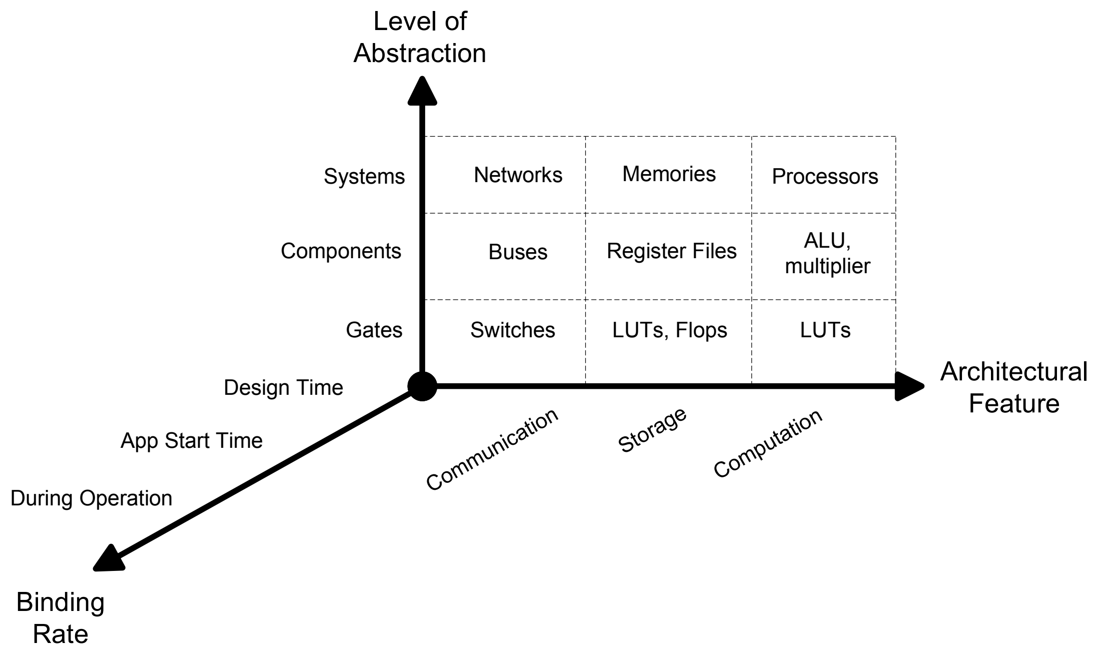


Figure 2.3: Verbauwhede and Schaumont's taxonomy of reconfiguration (from [1]).

In addition to categorizing RTR applications on the basis of binding rate, abstraction level, and architectural features, a distinction can be made based on when the configuration files, called bitstreams, were created. Just-In-Time (JIT) customization involves creating or modifying a circuit during operation, tailoring the design to the specific conditions [18] [19]. Virtual hardware, on the other hand, uses the dynamic reconfigurability of an FPGA to emulate a much larger device, much like virtual memory in a computer [20] [21]. At any instance in time, only a portion of the entire design is resident and functioning on the device.

A common optimization for JIT customization is constant propagation, also known as data folding, wherein constants that are only known at runtime, such as filter coefficients and encryption keys, are hard-coded directly into the hardware, reducing area and improving performance [19]. For many applications, however, the slow speed of FPGA implementation tools limit the utility of JIT customization as the traditional implementation flow requires seconds to hours to complete. Alternative tool flows exist [22], but these tools suffer several limitations. Architectural modifications to FPGAs have been considered to simplify the implementation process and speed JIT compilation [23], though currently these are strictly

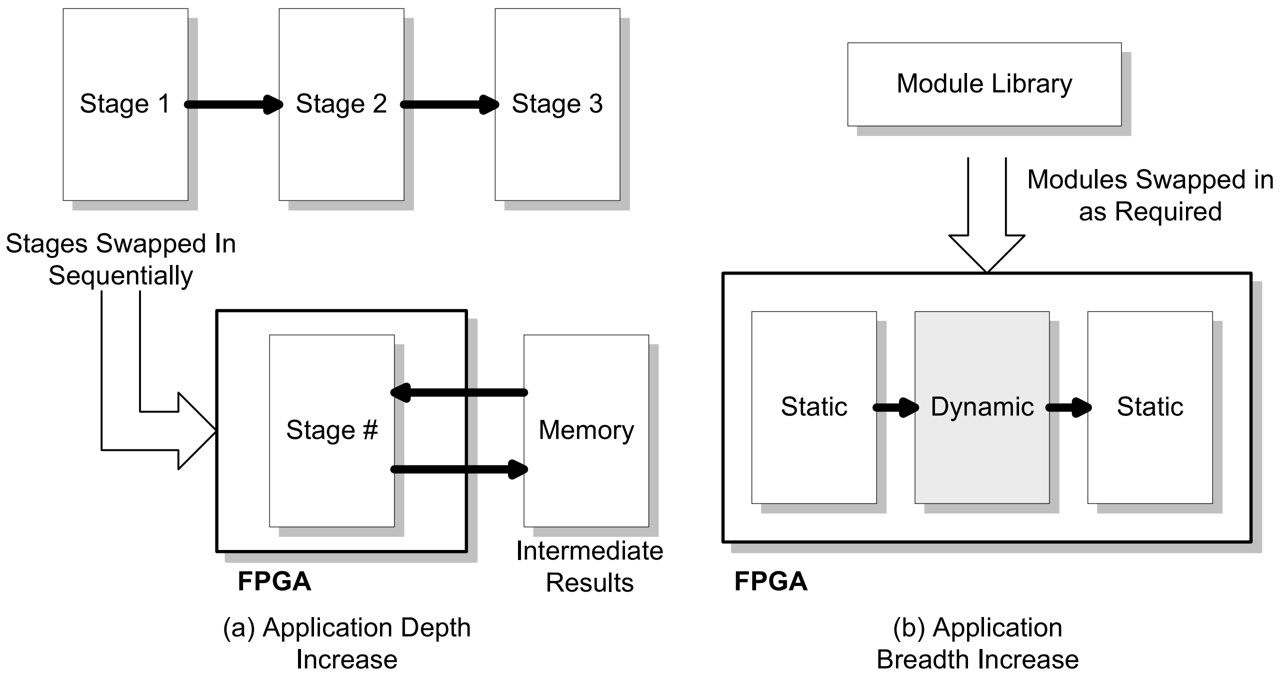

Stage 1
Stage 2
Stage 3
Stages Swapped In Sequentially
Stage #
Memory
Intermediate Results
FPGA
(a) Application Depth Increase
Module Library
Modules Swapped in as Required
Static
Dynamic
Static
FPGA
(b) Application Breadth Increase

Figure 2.4: Virtual hardware for increasing an application's breadth (a) and depth (b).

academic in nature.

Virtual hardware may be used to increase the depth of a sequential pipeline, by swapping pipeline stages in and out of the device, or to increase the breath of a design, by swapping in and out functionality as required. Figure 2.4 demonstrates the difference between RTR for breadth and RTR for depth. When an entire design is not resident on the device at a single time, as is the case when pipeline stages are swapped in and out of the device, some form of data buffering must occur between stages.

Numerous research projects have incorporated RTR into their designs. Common applications include automatic target recognition [24], gene sequencing [2], image and video processing [25], network applications [3], electronic design automation [26], neural networks [4], and instruction set extension [27]. While this list validates the utility of RTR in a variety of applications, it is quite small considering the length of time that RTR has been considered. If suitable tools and techniques existed the list of RTR applications could be much larger.

In general, candidates for virtual hardware are applications that can be divided into distinct sequential stages or applications in which certain circuits are mutually exclusive. Applications utilizing constants that are defined only at run-time (encryption, gene sequencing, filters, etc.) are candidates for JIT customization.

One of the more impressive results of JIT customization is a DES encryption implementation described by Patterson [28]. The encryption circuit is customized at runtime to the specific encryption key, resulting in a throughput that is greater than a generic ASIC implementation. The encryption circuit was tailored to its key using JBits [22]. A Java API to the configuration bitstreams of certain families of Xilinx FPGAs, JBits permits the designer to quickly modify bitstreams at the lowest level. A JBits Java program, when executed, produces a bitstream without running the traditional implementation flow. The fast implementation time, combined with inherent support for runtime parameterizable cores, makes JBits a powerful tool for JIT customization.

Additional projects utilizing JIT customization include neural networks [4] [29], where constant coefficient multipliers are updated at run-time, gene sequencing [2], with the search sequence defined at run-time, and boolean satisfiability [26], where the specific problem is not known at design time. In a gene sequencing application, Lemoine [2] was able to achieve a two to three order of magnitude speed-up compared to a processor, even with the JIT synthesis overhead of running the entire implementation toolchain.

An an alternative to running the entire implementation tool flow, several projects make use of JBits for directly generating a bitstream and reducing the overhead of JIT compilation. However there are severe limitations to the power of JBits and JIT compilation. JBits only supports a subset of older Xilinx FPGA families. Timing information is not available from within JBits, making timing-driven placement and routing impossible. While a recent project aimed to alleviate certain limitations by fusing the JHDL design environment with JBits, the JHDLBits project was abandoned before usable tools were released [30]. Even assuming these issues are addressed, the much shorter implementation times of JIT compilation will always produce an inferior result when compared to longer running traditional implementation tools. Because of these issues, JIT compilation is not a focus of this project.

FPGA configuration files may be used to reconfigure the entire device, a process known as full reconfiguration, or may only reconfigure a section of the hardware through partial reconfiguration. Full reconfiguration has the benefit of using the traditional implementation tools to generate the configuration files. However, fully reconfiguring the device may take a significant amount of time, potentially on the order of tens of milliseconds, during which the device may be unavailable. Also, without a large design effort, full reconfiguration requires an external host for control. Partial reconfiguration, on the other hand, permits sections of the FPGA that are not being reconfigured to continue functioning. Because of this, a partially reconfigurable design may implement its own configuration controller internally [31]. Finally, as only sections of the device are being reconfigured, partial reconfiguration reduces the latency of dynamically configuring the FPGA.

As previously mentioned, virtual hardware can be used to increase an application's breadth or its depth. One of the first uses of virtual hardware to increase an application's depth was RRANN, a neural network project that divided the training task into three sequential stages [32]. By using RTR the final project required only a single FPGA instead of three. The overhead of fully reconfiguring the FPGA hurt performance, prompting the authors' to rework the design to utilize partial reconfiguration [33]. An RTR design by Villasenor similarly decomposes video compression into three stages: discrete wavelet transform, quantization, and entropy encoding [34]. By executing each stage sequentially on the FPGA, the required resources are reduced by a factor of three.

An alternative use for virtual hardware is to increase an application's breadth. In these applications there is a complete application in the FPGA at all times. As required, new circuits may replace existing functions on the FPGA. Software Defined Radio (SDR) may be seen as an example of this. At any given moment the FPGA may implement a specific radio modulation scheme. When requested, the modulator in the FPGA can be replaced from a library of existing, stored modulators. The modulator designs are mutually exclusive, in that only a single modulator will be running at any one time. Increasing an application's breadth may reduce the effects of reconfiguration overhead as, for many domains, the rate of reconfiguration is less than for utilizing virtual hardware to increase an application's depth. This is the case for the commercial FALCON II radio marketed by Harris Corp., featuring software-controlled reconfiguration of its hardware [35].

An additional virtual hardware domain that increases the breadth of computation is that of instruction set extension. Several research projects have tightly coupled a configurable fabric with a processor for the purpose of adding custom instructions tailored to a given application [36] [37] [27]. An incarnation of this technology is currently marketed by Stretch, Inc. [38].

2.2 Design Methodologies

While design complexity increases with time in any domain, the manufacturing process improvements characterized by Moore's Law have doubled transistor density every 18 months. The first integrated circuit designs were completely hand crafted, limiting devices to thousands of transistors. Over time tools and methodologies were developed, permitting the designer to operate at higher levels of abstraction and increasing his or her design efficiency. Instead of transistors or logic gates, designers now operate at the level of registers, on the lower end of the abstraction spectrum, and existing blocks of intellectual property, on the high end. A few lines of code in a Hardware Description Language (HDL) can describe a design incorporating thousands of transistors.

In spite of these advances, there is a growing design-productivity gap. The number of transistors available to a designer is growing much faster than a designer's ability to effectively use them [39]. With many computer chips now containing hundreds of millions of transistors, much research in industry and academia is focused on automating time consuming design steps, such as Hardware / Software partitioning [40] and Register Transfer Level (RTL) design [41], permitting the designer to work at a higher level of abstraction.

Hardware designs start from a high level specification expressing function and performance requirements. Generally a functional model is then created in a High Level Language (HLL), such as C or MATLAB. Subsequent design iterations can then be compared to this high level model to ensure correctness. For the vast majority of hardware, the final design is expressed at the RTL level in an HDL, describing the flow of data between registers. This RTL description is at a much lower level than the initial HLL description and significant effort is required to translate the specifications expressed in the HLL to RTL HDL. Thus, in essence, the application has been described twice, once at a high-level and once at a low-level.

To further increase the level of abstraction, and thus a designer's efficiency, High Level Synthesis (HLS) has been suggested by many [42]. HLS involves an automated translation of a behavioral description expressed in an HLL into a description appropriate for hardware implementation, usually an HDL. These researchers feel that HLS is the natural progression of design automation. The time consuming tasks of control and datapath definition, datapath sizing, pipelining, etc., normally performed by experienced and costly hardware design engineers, are replaced by computer programs driven by optimization routines and heuristics.

The Processor Reconfiguration through Instruction-Set Metamorphosis (PRISM) project was one of the first attempts at automatic generation of FPGA accelerator cores from a high-level specification [37]. A traditional C program, decomposed by the programmer into functions, is analyzed and functions suitable for hardware implementation are automatically identified. A C-to-gates flow then produces hardware accelerators for these simple functions, which are then integrated back into the executable C code. During program execution the processor, when encountering an accelerated function, writes the operands to the FPGA coprocessor which produces results within a single cycle. This initial attempt at high-level synthesis limits the subset of C that can be accelerated and imposes the requirement that all generated hardware produce a result in a single cycle.

Many other academic HLS projects followed PRISM, including Streams-C, a project out of Los Alamos National Laboratory that produces synthesizable HDL from a subset of ANSI C [43]. Utilizing the Communicating Sequential Processes (CSP) computational and communication model, the user describes his or her application as a set of concurrently running processes communicating through blocking data streams. This model is well suited to streaming applications such as multimedia, DSP, and cryptography. The compiler, a modification of Stanfords SUIF work, synthesizes for each process a Finite State Machine (FSM) controller and a pipelined datapath. While compiler inefficiencies reduce performance by a factor of two to three over handcrafted designs, the order of magnitude productivity increase could justify its use for certain applications. Unlike PRISM, Streams-C generates a stand-alone application without the requirement of a host processor.

Several researchers have turned to SystemC to provide a simulation environment for RTR [44] [45]. These projects are very high-level in nature, simulating the design at the transaction level. This Transaction Level Modeling focuses on the communication between modules, abstracting away the modules' implementations. As such, this approach provides no support for evaluating different design implementations.

In addition to programming languages, other projects use a model-based approach. Representative of these is the University of Tennessee's CHAMPION framework, targeting image processing applications [46]. Dataflow-based applications are constructed by connecting pre-defined modules, each module containing C++ and VHDL descriptions permitting high-level simulation in software before implementation. The design is automatically partitioned across multi-FPGA CCMs. Although tailored to image processing there is no reason why this method, graphically connecting modules from a predefined library, could not be applied to other domains.

The few projects discussed above are representative of the many academic HLS tools that have been created. However, HLS is no longer just a topic of research. Several companies currently offer commercial quality HLS tools. Supporting C-based design is Celoxica's Handel-C [47], Impulse Accelerated Technologies' Impulse C [48], and Nallatech's DIME-C [49]. SRC Computer's CARTE design environment generates hardware from a C or FORTRAN description [50]. Model-based design tools, marketed toward DSP applications, are being offered by Xilinx with its System Generator [51].

HLS is not without its critics, who point to the suboptimal designs HLS produces compared to experienced hardware engineers and the inappropriateness of sequential programming models for describing parallel hardware. However, HLS significantly reduces design time and costs. It should be noted that any design automation produces inefficiencies. Programming at the assembly level or constructing circuits manually out of transistors produce better performing designs than coding with C or Verilog, respectively. Just as industry has accepted HLL compiler inefficiencies for software development, in many circumstances the performance penalty of these hardware design tools may be worth the reduced design costs and faster time-to-market. Furthermore, while not definitive, a recent comparison of several HLL-to-gates compilers indicated that for some applications the performance penalty of HLS is marginal compared to standard HDLs [52].

With few exceptions, existing design methodologies do not support dynamic hardware. This is due, in part, to the ASIC heritage of FPGA design tools. The high manufacturing costs and fixed structure of ASIC designs led to the development of robust design tools, as an ASIC implementation must function correctly on the first attempt. Reconfigurable computing has benefited from ASIC tool development, borrowing many of the ideas and algorithms. However, the implicit assumption in the ASIC world of static hardware hinders the development of RTR applications using the existing ASIC-based tools and models.

Traditional hardware design flows lack basic constructs and tools required for RTR application development, including:

- Methods for specifying dynamic communication and computational structures. Dynamically reconfigurable hardware places additional demands on any communication structure. Tradition design methodologies push this added complexity back onto the designer, further complicating the design process.

- Simulation of dynamic hardware. All commercial hardware simulators implicitly assume that the hardware is static, requiring any verification of the dynamic-nature of the application to be performed in hardware. This lengthens the design cycle and increases the probability of design errors in a released product.
- Design abstractions for reconfiguration. While synthesizable HDL may be easily ported from one configurable architecture to another, configuration interfaces vary across architectures, necessitating redesign.

To address the difficulties in applying traditional design methodologies to RTR applications several researchers have proposed or implemented new methodologies targeting the requirements of dynamic hardware.

Janus [5] was an early effort at a unified RTR application development environment centered around Java. Software for the host PC was written in Java while the hardware for the multi-FPGA system was created from JHDL, a Java-based structural hardware description language. JHDL was chosen after a previous attempt at a high-level language, GDL, encountered difficulty in solving multiple problems associated with HLL-to-gates synthesis [53]. Using the same environment for both hardware and software, Janus speeds development and enables high-level simulation of hardware / software interaction. A configuration controller residing on the host PC is automatically generated, managing the configurations of each FPGA in the system.

Janus was created under the coprocessor paradigm in which the FPGA is essentially a slave to an external host processor. Partial reconfiguration and dynamic scheduling are not supported. For the attached coprocessor computational model the lack of partial reconfiguration is not a limitation, although for embedded designs this omission would be an obstacle. Janus did not gain acceptance owing largely to the choice of seldom-used and low-level JHDL as a design language.

Eisenring and Platzner's RTR Framework [54] describes a tool-independent design and implementation methodology. The design is specified by a problem graph, an architecture graph, and a mapping between the two. This formalism simplifies tool development. The synchronous dataflow examples provided utilize three nodes types for design capture: task, buffer, and dispatcher. Hierarchical configuration control is achieved through a separate configurator node running on the host processor. The architecture graph specifies the target device and may include processors, memories, buses, and FPGAs. A set of constraints guide the allowable architecture graph to problem graph mappings. Like Janus, partial reconfiguration is not supported and a host processor is required. As years have passed since the initial description and no implementation has yet been described, it appears that progress on this framework has ceased.

The PADReH framework [55] focuses solely on hardware development, defining an open development flow permitting multiple methods of design capture, simulation, and partitioning to be used. Partial bitstream generation occurs within the Xilinx Modular Design Flow, which is the only fully specified step in the framework. An example of a configurable instruction set processor was created. Little is provided to the designer in terms of tools or abstractions.

Berkley's Stream Computations Organized for Reconfigurable Execution (SCORE) project [56] proposes an FPGA-like architecture. Multiple fine-grained reconfigurable regions on the chip communicate through large First-In-First-Out (FIFO) buffers implemented in memory. An on-chip processor can run traditional programs while managing the scheduling of the reconfigurable arrays. Hardware pages can be swapped in and out of these regions dynamically. Unlike an FPGA, these pages are location-independent. Applications, consisting of a datapath and FSM, are described using TDF, an RTL-like hardware description language created specifically for SCORE. Designed for extending an application's depth by swapping sequential pipeline stages in and out of hardware, SCORE can also extend an application's breath. In spite of the suitability of SCORE for streaming applications, no commercially available devices exist.

Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems (SPARCS) [6] starts with a behavioral VHDL description of the application separated into tasks communicating through shared memory or direct connections. Temporal and spatial scheduling occurs across multiple FPGAs. A high-level synthesis tool converts the behavioral description to RTL that is then processed with traditional tools.

The Caronte PR framework defines a high-level development environment targeting co-processor applications [57]. Simulation of PR occurs is possible via SystemC, with design entry via HDLs or Impulse C [58]. Caronte's use of Impulse C differs from the work presented in this paper in that Caronte merely uses Impulse C to produce HDL and not capture the totality of the application including the configuration control. The bus-based communication of Caronte limits its applicability to streaming applications.

The Institute for Software Integrated Systems (ISIS) describes a prototype model-integrated design environment for dataflow applications [59]. ISIS focuses on constraint-driven development and verification. Tools automatically apply user-specified constraints to prune the design space. Co-simulation is provided for at multiple levels of abstraction. A complete runtime environment is described linking the dynamic hardware modules to a software OS. Design entry occurs via graphical tools linking to pre-described modules. The development environment targets board-level designs comprised of heterogeneous computing elements (FPGAs, DSPs, processors, etc.), limiting the utility for FPGA-centric applications. Partial reconfiguration is not supported.

Luk et al. from Imperial College [60] describe a framework of tools supporting RTR for the Xilinx XC6200 FPGA. The unique architecture of the XC6200 limits the applicability of their work to modern FPGAs. Their modeling methodology involves using multiplexers to select which module from a set is active at any one time. The multiplexer select lines are then controlled by a FSM. If space is available on the FPGA, no reconfiguration is required, with the multiplexers selecting which module is active.

More recent work from Imperial College is of particular relevance to this research. By defining abstractions of low-level details, a HLL-based approach to RTR application development is described [61]. A modified form of C, RT-C, captures the design behaviour at a high-level, including configuration control. The RT-C is then translated into Handel-C [47], a commercial C-to-gates synthesis tool. An implementation flow generates the required configuration files, with configuration management handled by a host processor. The implementation flow, however, is based on JBits and therefore is limited to older architectures. Also, a manual translation is required to go from the Handel-C generated HDL to JBits and the resulting design is shackled to a host processor.

Brigham Young University developed a JHDL-based Reconfigurable Computing Application Framework (RCAF) with the distinguishing feature that the framework, consisting of control, communication, and debugging aids, is deployed in the finished product [62]. The framework assumes a tight integration of the FPGA with a host processor running a controlling Java program. While an excellent debugging and I/O platform, this framework does little to facilitate the capture of configuration management or the incorporation of embedded processors.

These previous projects, summarized in Table 2.1, all suffer from the major omission of partial reconfiguration support. Additionally, most assume a model of external configuration control, mandating the use of a host processor. For embedded application this requirement is generally prohibitive. It is also interesting to note that no project has been extended, by its authors or others, since its initial implementation. This is perhaps in part due to the tight coupling of many of these frameworks to a specific architecture or design capture tool.

Project	**Design Entry**	**Model of Computation**	**Architecture**	**Limitations**
Janus	JHDL	unspecified	host + FPGA	No partial RTR Requires host
SCORE	modified C	dataflow	custom	Custom architecture
SPARCS	behavioral HDL	dataflow	host + FPGA	Requires macro library No partial RTR
Eisenring's	dataflow graph	dataflow	host + FPGA	Incomplete description No partial RTR
Model-Integrated	dataflow graph	dataflow	independent	No partial RTR Requires model library
RCAF	JHDL	unspecified	host + FPGA	No partial RTR Requires host Few abstractions
Imperial College	library-based	dataflow	XC6200	Few abstractions
Imperial College	RT-C	dataflow	limited by JBits	Requires host Manual translation
Caronte	various	co-processor	embedded proc	Limited automation
SystemC	SystemC	unspecified	unspecified	Simulaton only

Table 2.1: Previous RTR Development Environments

Chapter 3

Approach

The central contributions of this dissertation are tools and techniques for simplifying dynamic hardware development. To this end, design abstractions supporting dynamic hardware in streaming applications have been defined. Leveraging these models, tools have been developed that significantly reduce a dynamic hardware designer's workload. Novel features discussed in this chapter include the incorporation of partial reconfiguration, on-chip configuration management, seamless integration of embedded processors, and a mechanism to guarantee equivalence of high-level simulation and hardware execution.

The complete approach consists of an architecture-agnostic frontend design flow coupled with an architecture-specific backend implementation flow. As shown in Figure 3.1, the inputs to the design flow are a functional description of the application in a high-level language. The design flow translate this specification into HDL and software along with a special RTR Computing Specification Format (RCSF) file. All design flow outputs are passed to the backend flow, the output of which are the required configuration bitstreams for the FPGA. The interface between the design and implementation flows is neutral from an architectural and design environment standpoint, permitting the direct porting of the designs between architectures.

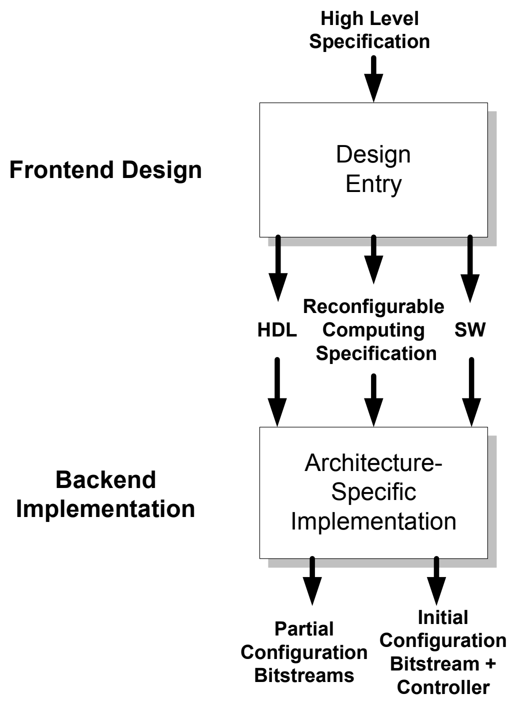

High Level Specification
Frontend Design
Design Entry
HDL
Reconfigurable Computing Specification
SW
Backend Implementation
Architecture-Specific Implementation
Partial Configuration Bitstreams
Initial Configuration Bitstream + Controller

Figure 3.1: Combined design and implementation flow.

This chapter first describes the models used to abstract computation, communication, programming, and reconfiguration. Next, an explanation of the architecture-agnostic frontend design flow is presented, followed by a description of the architecture-specific backend implementation flow.

3.1 Models and Abstractions

Models and abstractions are important in hardware design, simplifying the design process by limiting the design space, hiding low-level details, and facilitating verification. There is a trade-off between the flexibility of a model and the design time. Less restrictive models provide the designer with greater flexibility at the expense of increased design time to prune the larger design space. For configurable computing, the models chosen for computation, communication, and reconfiguration greatly affect the design difficulty.

A model of computation describes how computational elements are constructed and operate. Common models of computation include Finite State Machines (FSMs), continuous time, multi-threaded, and dataflow. Ideally the model should be selected to fit the application. For example, FSMs are well suited for use as controllers while dataflow computation accurately describes many signal processing problems. Models may be mixed; a datapath controlled by an FSM is a commonly used computational model for hardware.

Communication models specify how data and control signals are exchanged between design elements. For multi-threaded programming, common communication models are message passing and shared memory. Hardware modules may communicate via direct connections, buses, shared memory, and networks, among other methods. An improper selection of the communication model can significantly impact performance.

The model of reconfiguration describes how reconfiguration is managed. Several previous projects controlled FPGA configuration via an attached host processor. Other methods have included hierarchical FSMs and network reconfiguration. The reconfiguration model affects the types of applications that can benefit from RTR. For example, processor-controlled reconfiguration permits complex configuration schedules that may tailor the circuits to the environment at a much finer level than an FSM controller. However, this scheduling flexibility comes at the expense of a processor.

The term *programming model* generally is applied only to software-programmable processors to describe the mechanisms and abstractions of which the programmer may make use. The programming model describes how components (processors, threads, I/O, etc.) interact from the programmer's point of view. As this dissertation fully integrates embedded processors into RTR design, a programming model is required to interface these processors with the dynamic hardware.

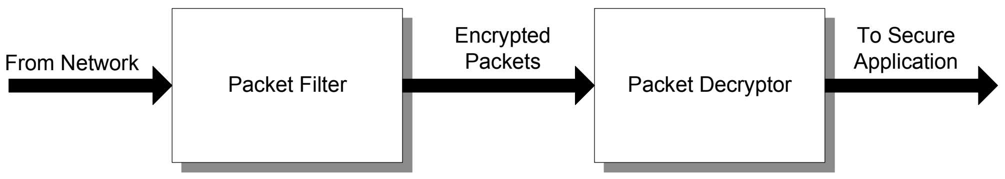


Figure 3.2: Secure network streaming application.

3.1.1 Computation and Communication Models

The models of computation and communication were selected to favor the traditional applications of FPGAs, namely streaming applications. Streaming applications, consisting of a repeatable schedule of computations operating on a steady flow of data, are typically found in networking, signal processing, and cryptographic domains, all strong suits of configurable logic. These applications generally benefit from low-overhead, high bandwidth communication channels and deep computational pipelines. Streaming applications are typically decomposed into a pipeline of independent computational elements connected by unidirectional data streams. These computational elements function concurrently to one another, sharing no state or information except the data that is passed through streams. An example of a streaming application is an encrypted network interface, a diagram of which is shown in Figure 3.2. Packets from the network are first filtered with the encrypted payload streamed to a decryption unit.

The focus on streaming applications limits the utility of this research to other arenas, such as High Performance Computing (HPC) applications. Although there has been a resurgence of interest in FPGAs in HPC, an economic analysis indicates that FPGAs are not yet cost competitive with commodity processors for the floating point applications typical in HPC [63].

Several computational and communication models can accurately describe streaming applications, including several dataflow flow models and the CSP model. In selecting an appropriate model it was imperative that the actual functionality of hardware be captured. It is desired that the model of computation be accepted by the design community, as demonstrated by the availability of commercial development tools. An additional desirable trait is that of determinism. For a fixed stream of input data, the output of the application should be the identical regardless of the execution platform.

Kahn Process Networks (KPNs) are commonly used in DSP development environments [64]. A KPN is a collection of concurrently executing processes that communicate via infinitely long unidirectional FIFO buffers. A write operation is non-blocking and always succeeds while a read operation blocks until data is present in the FIFO. KPNs are, with a few qualifications, provably deterministic.

A somewhat related model of computation is CSP [65]. Like KPNs, concurrently running processes in CSP communicate over unidirectional FIFO streams. However, in CSP there is no notion of an infinitely long FIFO buffer, with write operations blocking when the finite storage in the FIFO buffer has been exhausted. Because of this, CSP more accurately models hardware. With some qualifications, such as an infinite FIFO stream buffer, CSP can be made equivalent to KPN.

Several development environments and languages support the CSP model, such as the CoDeveloper toolset [48], the Occam programming language [66], Codetronix Mobius development environment [67], and the FDR2 refinement checker [68]. Additionally, with few exceptions, the many products that support dataflow models of computation such as KPN can be utilized within the CSP model, including the Ptolomey modeling tools [69]. Traditional HDL design tools can also easily capture and implement streaming applications using the CSP model, as has been demonstrated by the creation of CSP macros for Verilog [70]. For these reasons, CSP has been selected as the computational model for this dissertation.

CSP, as originally conceived, includes operators to describe non-deterministic behavior. For the purposes of this dissertation these operators are not allowed. Determinism is a greatly desired attribute, as it guarantees that the application will produce identical output across various execution platforms; ensuring, for example, that the software simulation matches the hardware implementation.

The implementation of the CSP application description is straightforward. Communication channels, or streams, can be created out of asynchronous FIFO buffers. These provide a high-bandwidth, low-latency connection between concurrent processes with minimal communication overhead. By using asynchronous FIFOs, processes can be clocked at different rates, potentially increasing the overall throughput. The FIFO-based communication permits easy integration with Xilinx embedded processors as both the Xilinx MicroBlaze soft processor [71] and later versions of the PowerPC processor found on Xilinx FPGAs feature Fast Simplex Link (FSL) interfaces that are nothing more than asynchronous FIFO buffers linking the processor to peripherals.

3.1.2 Reconfiguration Model

A methodology for configuration control has been selected that is centered on the idea of mutually exclusive modules. In this discussion it is assumed that a module is the hardware implementation of a single process in the CSP model. The designer identifies a set of modules that are mutually exclusive in that only one of the set's members is active in hardware at any one time, as shown in Figure 3.3. The figure describes a cryptographic application. The user may need to decrypt data from a network interface using a variety of algorithms. Leveraging the reconfigurability of the hardware, there is no need to implement in hardware every possible algorithm that may be required. The figure shows a set of these mutually exclusive cryptographic cores, of which only one will be resident in the dynamic hardware at a time. Any module within this set may be selected for implementation, at which time the configuration manager reconfigures the FPGA to swap in the selected module. During reconfiguration modules reading from or writing to the process being swapped into hardware will block until configuration is complete. This abstraction is similar the Swappable Logic Unit of Brebner [20] and the dynamic hardware modeling scheme of Luk [21].

The FIFO-based communication model reduces the importance of placement on communication throughput. Increased communication latency resulting from a poor placement of

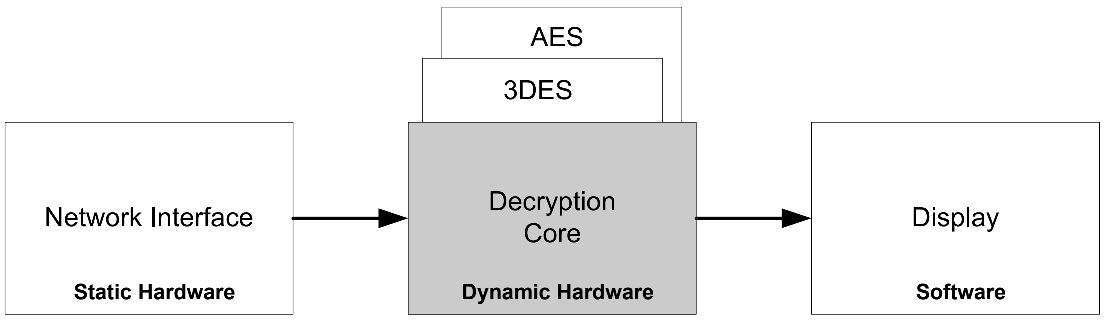


Figure 3.3: Mutually exclusive set of processes.

modules may be addressed by inserting additional storage elements in the communication streams. This increases the throughput by introducing addition cycles of latency, mitigating the effects of a sub-optimal placement and permitting research-quality tools for module placement to be utilized without a significant performance penalty.

This reconfiguration model enables the designer to utilize partial RTR to extend an application's breadth, by adding new functionality at runtime, or to extend an application's depth, by swapping pipelined application stages in and out of the device. It is left to the designer to properly buffer results between the application stages.

3.1.3 Programming Model

Embedded software has become a integral part of many systems. For FPGA-based designs embedded processors may perform critical control functions, interfacing the custom hardware with the outside environment. While several previous RTR development environments tightly integrate hardware and software design, these projects target systems with a dedicated host processor separate from the FPGA. This system model is unable to address embedded systems, where a separate host running a desktop operating system is not practical.

In this dissertation the programming model tightly integrates software with the dynamic hardware through a low-latency message passing interface. The processor may directly communicate with the dynamic hardware via reads and writes to the FIFO-based data streams.

This approach, adopted by Williams and Bergman [72] in their uCLinux port to the MicroBlaze soft processor, fully integrates processors into the CSP model. While this model limits interactions between hardware and the processor to the passing of data, for the streaming applications targeted in this dissertation the hardware control overhead is minimal with the majority of communication between the processor and hardware being data-related.

In the event that a different communication model between hardware and software better suits the application, the designer may break with the CSP model and utilize shared memory or dedicated control signals.

3.2 Design Flow

The high-level design flow, shown in Figure 3.4, consists of a front-end high-level design entry and synthesis environment accepting an HLL application description. High-level synthesis techniques, using any development environment supporting the CSP computational model, produce synthesizable HDL for implementation by the architecture-specific backend flow. The inherit flexibility in the methodology permits the specific development environment to be chosen to suit the application. For HPC applications an HLL-based environment, such as the Impulse C-based CoDeveloper tools, may be used. For embedded, timing-critical applications a library-based approach, such as found in Xilinx System Generator, may be appropriate.

Though the development environment is not specified, the methodology does stipulate that the environment permit high-level simulation and hardware / software partitioning. Unlike some previous attempts, HW / SW partitioning is performed under user control, as commercial-quality tools automating this partitioning are unavailable. When partitioning methods are sufficiently mature they may be easily integrated into the framework. The application developer may utilize profiling tools to identify critical tasks for implementation in hardware.

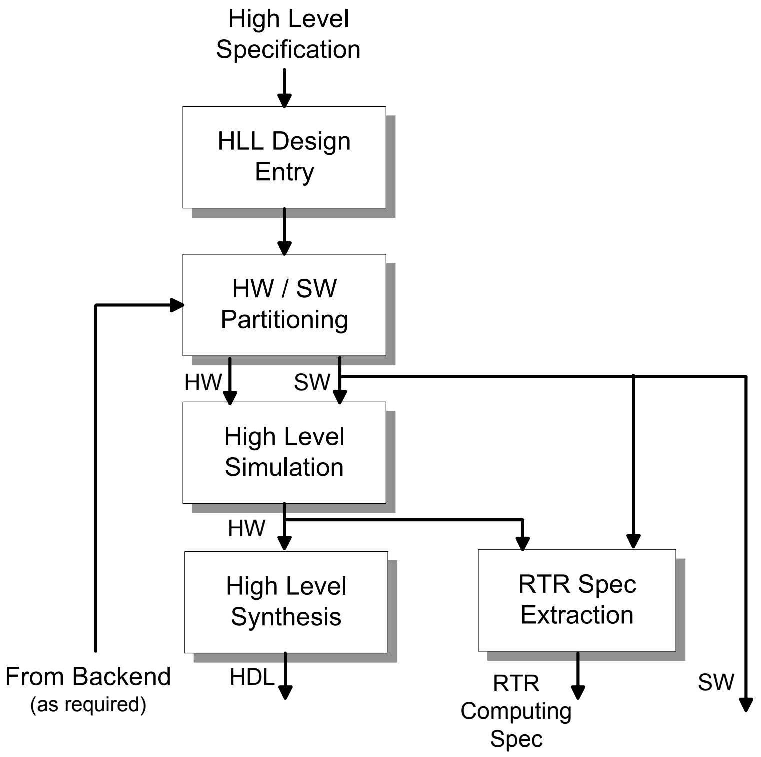


Figure 3.4: Design flow.

After partitioning, a high-level simulation, discussed below, is performed. The design is then compiled to synthesizable HDL by the high-level synthesis tools.

3.2.1 Design Entry and Partitioning

The frontend design flow makes use of commercial-quality high-level development tools for design entry. While this work utilizes Impulse C [48], any high-level development environment supporting the CSP model may be used, including AccelDSP [73] and System Generator [51]. Regardless of the specific environment used, the procedure is the same for design entry.

Design entry begins by partitioning the high-level specifications into separate modules. For the streaming applications targeted by this dissertation divisions between modules can occur at the natural boundaries between different computations. This stage permits the designer to identify parallelism and concurrency in the design. While other projects automate this partitioning, a designer familiar with the application is generally better at extracting high-level, coarse-grained parallelism than current algorithms. The high-speed, low-latency FIFO-based communication between modules simplify this partitioning and permit the designer to easily repartition without redesigning a communication scheme. Similarly, the CSP communication model ensures correct synchronization between the modules regardless of the chosen partitioning.

Hardware / Software partitioning is performed under direct designer control. In spite of years of research [40], no commercially successful automated partitioning tools exist; though tools, such as System C, do exist to investigate the effects of different partitioning schemes. Streaming applications, unlike HPC applications, can be straightforward to partition. The computational datapath is generally placed in hardware with control and interface functions placed in software. For applications where the division between hardware and software is less apparent, any software profiling tool may be used to assist the designer in locating code appropriate for hardware implementation.

While a high-level design specification greatly simplifies application development, it also reduces performance compared to hand-crafted hardware. For those cases where performance is paramount, the hardware generated by the HLS tools may be augmented with hand-crafted code in the implementation phase, with a simple behavioral model of the custom hardware utilized for high-level simulation. The RTR Control Specification file can be easily edited to add HDL or netlists generated from other sources.

3.2.2 Simulation

High-level simulation of an integrated HW / SW RTR application is an ability found only in relatively few research projects, notably Janus [5]. By simulating the entire design early in the design cycle functionality can be quickly verified and integration issues identified. Furthermore, behavioral simulations run much faster than gate-level simulations, permitting more thorough tests to be run.

The design flow specifies that high-level simulation be performed prior to HLS. Many commercially available design environments facilitate this simulation. In Impulse C, for example, simulation is performed by compiling the design with special simulation libraries. When the result is executed, each CSP process is started as a separate software thread communicating via blocking reads and writes to shared memory buffers.

In order for any simulation to be effective, the simulation must accurately model the real world. For the case of RTL simulations this is accomplished by simulating the hardware design at the register level. For high-level simulations, however, no hardware details are present and only the functionality is modeled. The lack of lower-level details prevents any high-level simulation from matching actual hardware on a clock cycle basis, as the simulator in this case knows nothing about the clock. The best that a high-level simulation can accomplish is to correctly model the system's output for a given input sequence.

3.3 Implementation Flow

RTR modifications to the frontend design flow enable high-level simulation of designs. However, a backend implementation flow is required to actually create the architecture-specific partial bitstreams required for RTR. Numerous previous projects have attempted to produce a usable RTR implementation flow with limited success. These projects generally have taken one of two forms: a modification of the Xilinx Modular Design Flow [74] or extension of the low-level JBits tools [22].

Design flows based on JBits, while significantly more powerful than those using the Modular Design Flow, are limited to the older Xilinx devices that JBits supports. Because JBits functions at such a low level, development time is significantly increased.

Projects that have targeted the Modular Design Flow have seen a limited lifetime as Xilinx's support for partial reconfiguration has varied significantly with each version of their implementation tools. Recent market trends towards SDR have prompted FPGA vendors to finally support RTR as an integral part of their tools, simplifying development of reconfigurable hardware [9]. To mitigate the effects of modifications to the configuration architecture or development tools, this dissertation clearly separates frontend design from backend implementation, permitting any RTR-capable backend implementation flow to be used with only minor modifications to parse the RCSF file.

The backend implementation flow, shown in Figure 3.5, accepts HDL and software from the frontend design phase. Commercial synthesis tools convert the HDL into a gate-level netlist. Based on the size of the resulting modules, the design is area constrained and floorplanned using tools developed for this dissertation. Vendor-supplied place and route and timing analysis tools are then run to determine the maximum clock frequency for each module. From these numbers and RTL simulation results, the throughput of the final design can be calculated. If this performance is unacceptable three options exist. The implementation flow can be repeated after reconstraining the design to provide critical modules with more area or better placement. Alternatively, additional tasks may be moved into hardware by repeating HW/SW partitioning in the frontend design flow. Finally, different implementations of the application, from an algorithmic level, may be attempted and the entire flow repeated. The extreme automation of this approach permits a quick design cycle, facilitating the comparison of a variety of implementations.

3.3.1 Reconfigurable Computing Specification Format

This methodology for capturing the design must include a format for specifying all aspects of the design. In traditional static hardware design, several formats may be used, depending on which stage of design is being performed. HDLs, such as Verilog and VHDL, capture the behavior of the design and the flow of data between registers. Other formats, such as EDIF, may be used to describe the gate-level netlists. Finally, an architecture-specific bitstream file

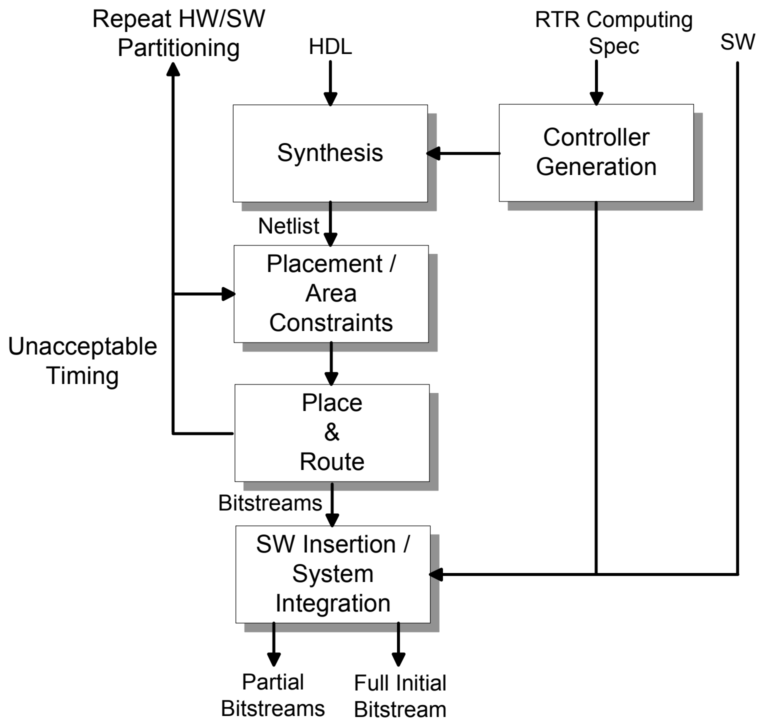


Figure 3.5: Architecture-specific implementation flow.

stores the configuration data for the device. Additional files are required for implementation. At the very least the design's inputs and outputs must be constrained to specific pins on the actual device.

For an RTR application, there are aspects to the design that cannot be specified easily using these traditional formats. The HDL-focused methodologies, with their ASIC roots, treat the hardware as static and include no provisions for describing dynamic modules or connections. The Xilinx partial reconfiguration flow [74] uses HDLs to capture the design of each module and the connections between them. To describe dynamic hardware the Xilinx implementation flow permits multiple modules, each with the same name and connections, to be created in such a way that there are interchangeable in the final design.

There are several severe limitations with the Xilinx approach. No format exists for capturing the list of dynamic modules. Configuration management is completely unspecified, with the designer forced to develop his or her own scheme. These omissions force designers to develop their own implementation methodologies and design capture formats, increasing design effort and inhibiting the exchange of designs. Finally, dynamic modification of wiring cannot be specified. As the Xilinx flow provides no method for modifying connections between modules, this is not a big limitation, though research is attempting to change this [75] [76] [77].

Previous projects have created their own methods for capturing the RTR-specific requirements. In general these formats have not been published or, where they are available, are not suited to other development environments. This dissertation addresses the deficiencies in existing work by developing a flexible file format, the RCSF, capturing the architecture-agnostic design details in a single, easy to understand format. The intention is to free the designer from a single development environment or device architecture. The RCSF will serve as the interface between the frontend design flow and the backend implementation flow. As the format should permit exchange of designs across different devices and architectures some board-level requirements will also be captured.

The following design information that is currently not captured in other files will be specified by the RCSF:

- List of dynamic modules, including connections.
- Location of files describing each module, along with the file format.
- List of embedded processors, including connections to other modules.
- Location of software for each processor.
- Architecture-independent configuration control information.

3.3.2 Configuration Management

A control mechanism is required to manage reconfiguration as none is present on an FPGA. This controller must determine when to reconfigure the device, fetch the appropriate bitstreams from external storage, and interface with the internal configuration port to perform the reconfiguration. During reconfiguration the logic under reconfiguration will be in an unknown state, potentially producing bogus outputs that may affect active logic. A mechanism for isolating the modules as they undergo reconfiguration must exist. For complex designs reconfiguration may need to be managed by software. As the configuration architecture varies across FPGA families, an architecture-independent method for describing configuration management is required to permit easy porting of applications.

3.4 Verification

Design verification is a well developed field for ASICs, where a single logic error can cost millions of dollars. A host of tools exist to verify the final design against the original specification, ensuring that the desired functionality was preserved by the tools.

In FPGAs design verification tools are not as widely used, as bugs found in hardware testing do not incur the large time and financial penalties that ASICs face. The addition of RTR and HLS add design dimensions that complicate traditional hardware testing and debugging. The benefits of this dissertation are lessened if a designer must spend hours debugging an ICAP controller or HLS-generated HDL.

While beyond the scope of this work, techniques have been identified to automate the verification of hardware created by this development environment. This include:

- Automatic creation of an HDL testbench from the designer's HLL testbench.
- Use of the configuration backplane to monitor the FIFO streams connecting modules. In the CSP model, the communication between processes is deterministic and synchronized, making them excellent points for comparing the hardware to simulated HDL and HLL results.
- Leverage and extension of tools from a related project [77] for verification of partial

bitstreams.

The goal of these techniques is to provide verification of the results of each step in the design flow to the previous one. These concepts are discussed more fully in Section 7.

3.5 Reconfiguration Synchronization

Designs based on the KPN and CSP computational domains can, with a few restrictions, be completely deterministic in nature. The rendezvous nature of communication provides synchronization in the absence of a clock. Determinism ensures that any simulation or implementation of the design will produce the same output for a given input. Unfortunately, the addition of reconfiguration can destroy this determinism if no mechanism exists to synchronize reconfiguration with communication.

Consider, for example, the system shown in Figure 3.6, consisting of an input stream that attaches, via a fork operation, to both a configuration controller and a reconfigurable set of operators. The fork module merely copies the single input stream to two output streams. Suppose the configuration controller is programmed to swap from the Left Shift process to the Right Shift module if the input exceeds a threshold of 255. Assume that the input is composed of the stream of numbers {2, 8, 256, 32}. The fork process presents this stream of numbers simultaneously to the controller and Left Shift which process new inputs in parallel.

Due to the asynchronous nature of computation in the CSP domain, there is no certainty which process, the controller or the left shift, will complete first. This leads to three possible system output streams.

If both processes read an input value with the controller analyzing the input and initiating a reconfiguration before Left Shift has computed its result, the system's output would be {4, 16, 16}. The input value of 256 would be lost when the Left Shift was removed from the hardware.

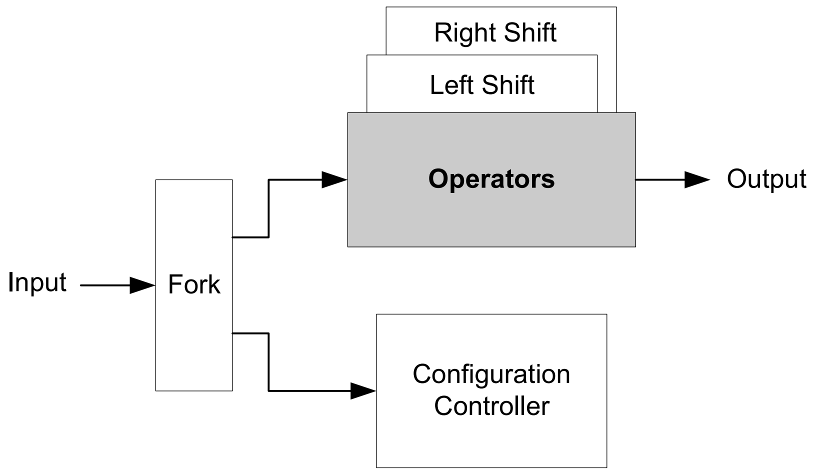


Figure 3.6: System with indeterminant output.

If, however, Left Shift operation completes prior to the controller initiating reconfiguration, the system's output would be {4, 16, 512, 16}. The input value of 256 would be doubled, possibly contrary to the designer's intent.

Finally, if the controller initiated reconfiguration before Left Shift read its data from the stream, the system's output would be {4, 16, 128, 16}. To complicate matters, this behavior could differ between runs, whether in hardware or software simulation.

It can be argued that for many applications this infrequent and brief indeterminacy is acceptable. In an SDR application the slight delay in reconfiguration or the loss of a small number of data samples might be acceptable for voice communications. However, in certain applications, such as network processing or cryptography, it is desirable to have guarantees concerning the correctness of the system's output at all times.

To provide determinacy, reconfiguration must be linked to communication, as in the CSP domain communication between processes is the only source of synchronization. Therefore to ensure determinacy in this scheme the configuration controller has a dedicated stream to each reconfigurable set, acting as a back channel for synchronizing processes. This is shown for the case of a single reconfigurable set in Figure 3.7.

The reconfigurable set reads from this *Reconfig* stream before it reads from its other

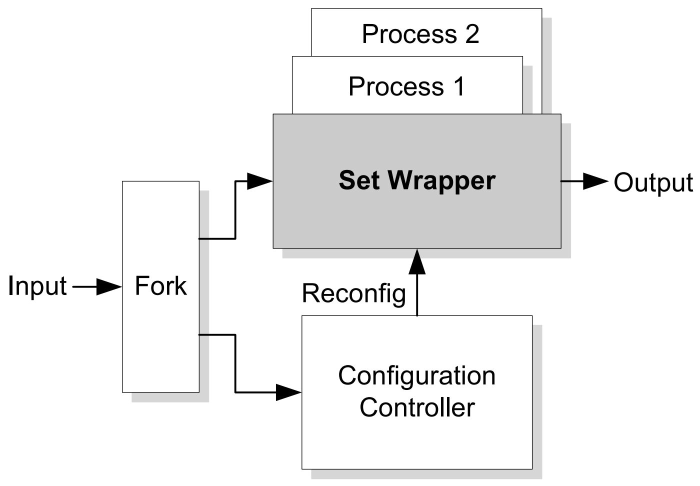


Figure 3.7: Synchronized system with deterministic output.

inputs. If a reconfiguration is required the set halts. The configuration controller blocks until the set has read from the *Reconfig* stream, reconfiguring the device as needed afterwards. This ensures that reconfiguration occurs repeatedly at the same time, relative to the incoming data stream, in both software simulation and hardware implementation.

The configuration controller will often reside on an embedded processor, taking several cycles to determine if a reconfiguration is required. If the controller must communicate a reconfiguration decision to the reconfigurable set with every input sample, the reconfigurable set will be forced to run at a much reduced rate. As the actual action of partially reconfiguring an FPGA takes a large number of clock cycles, the penalty for producing a reconfiguration decision every N input samples instead is minor. The actual value of N is set by the designer, possibly taking advantage of natural breaks in the input stream as would occur between packets in a packet-based system or frames in video processing.

To further reduce the data processing required by the controller, the *Fork* process in Figure 3.7 can reduce the rate at which data enters the controller by passing the controller a single sample for every L samples passed to the set wrapper. This data reduction can also represent the effect of preprocessing on the controller's data stream.

Though CSP can provide a valid description of the synchronization scheme, the use of

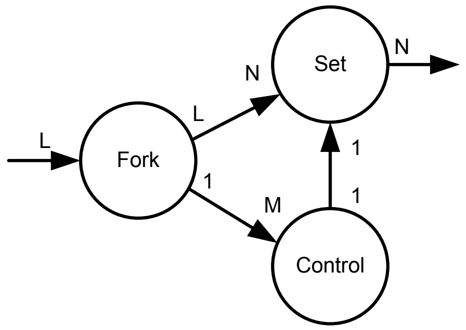


Figure 3.8: Synchronous Data Flow description of configuration control.

Synchronous Data Flow (SDF) techniques facilitate the derivation of the conditions necessary to prevent deadlock. In SDF, each process, or node, waits until all required data is available on its inputs before performing its computation, or firing [78].

Unlike CSP, the channels in SDF are considered to be FIFO buffers of limitless depth. To ensure that a physical implementation using fixed-depth buffers is possible, an analysis can be performed to verify that each process's sample rate is consistent. That is, the production and consumption rates of all processes in the circuit are such that the data residing in the buffers between processes does not grow with time. With consistent sample rates implementation is possible with limited-size FIFOs. This is equivalent to preventing deadlock in the CSP domain, as CSP processes block when any output channel's buffer is full.

Shown in Figure 3.8 is the synchronization scheme drawn as a data flow graph. The labels on each arc represent the number of data values, or tokens, produced or consumed at each firing of the node. There are three variables in this circuit:

- N, the number of tokens the set accepts before querying the configuration controller,
- M, the number of tokens the configuration controller accepts before communicating its reconfiguration decision, and
- L, the number of tokens consumed by a set for each data sample consumed by the controller.

The SDF graph of Figure 3.8 can be expressed as a topology matrix, with a row for each arc in the circuit and a column for each node. The (i, j)th entry in this matrix is the number of data tokens consumed (if negative), or produced (if positive) by node j on arc i at each firing of the node. For this synchronization circuit the topology matrix, T, is given by:

$$\begin{pmatrix} L & -N & 0 \\ 1 & 0 & -M \\ 0 & -1 & 1 \end{pmatrix}$$

In order for the sample rates of the nodes in this circuit to be consistent, it is necessary that the rank of the topology matrix T equal one less than the number of nodes [78]. The rank of a matrix is the number of linearly independent rows or columns. It is apparent by inspection that the second and third rows of T are independent. To ensure consistent sample rates, values for L, N, and M must be found such that the first row is a linear combination of the other two.

Considering each row as a column vector, a relationship between the three variables must be found such that:

$$c_1 \begin{pmatrix} 1 \\ 0 \\ -M \end{pmatrix} + c_2 \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} L \\ -N \\ 0 \end{pmatrix}$$

Readily apparent from the above equation, c_1 must equal L and c_2 must equal N. With these relations in hand, the problem becomes finding a relationship between L, M, and N that satisfies the bottom equation:

$$-M * c_1 + c_2 = 0$$

Inserting the values for c_1 and c_2 found above, the required relationship becomes evident.

$$-ML + N = 0$$

$$N = LM$$

When $N = LM$, the rank of the circuit's topology matrix is two, ensuring consistent sample rates and verifying the absence of deadlock in a physical implementation with CSP communication.

In this configuration scheme, any internal state, including pipeline state, of the reconfigurable process is lost when a reconfiguration is performed. For many applications this is likely to be acceptable but situations do exist, such as in cryptography, where the internal state of a process's pipeline should be recovered prior to reconfiguration.

To recover the data still in a process's pipeline before reconfiguration, counters are added to the set's inputs and outputs, as shown in Figure 3.9. When the control stream issues a reconfiguration command, the set's inputs are fed zeros until the pipeline is flushed. Logic added to the set determines when the flushing is complete by comparing the input counter to the output counter. For processes that produce an output for every input, these counters will differ by the pipeline depth. For more complex processes (cryptographic hash functions, decimating filters, etc.) that consume multiple input values before producing a single output, the designer specifies the ratio of input samples consumed to outputs produced. This information is used by the set's wrapper to determine when the process has completed flushing. Once the data in the pipeline has been completely flushed, the process's outputs are disabled in preparation for reconfiguration.

When a reconfiguration is indicated via the $Reconfig$ stream, the reconfigurable process is, in effect, disconnected from the incoming data stream, permitting the process to consume zeros as quickly as it can to flush its pipeline. Thus all synchronization between the set wrapper the configuration controller is lost. As the controller must wait until the reconfigurable process has flushed its state before performing a reconfiguration or risk losing pipeline

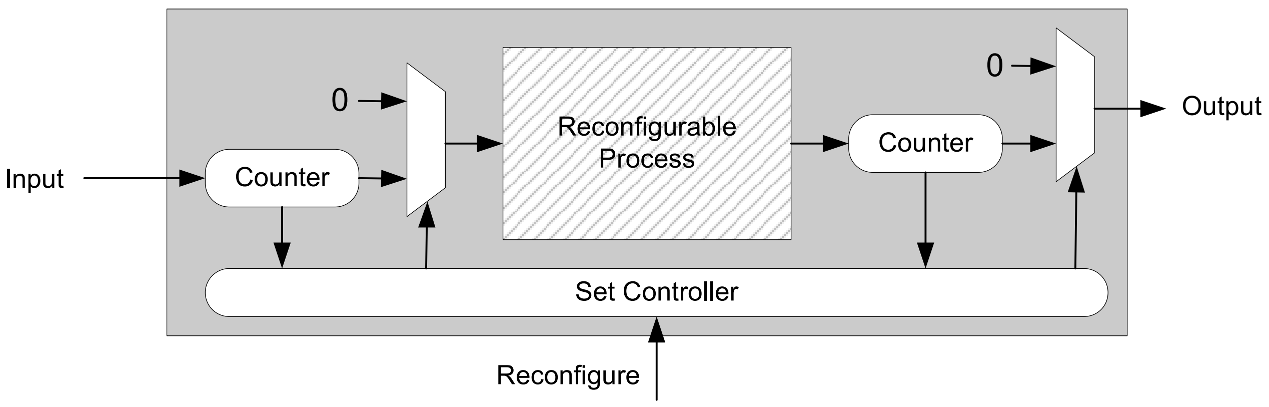


Figure 3.9: Pipeline flushing scheme.

state, some additional mechanism is required to notify the configuration controller that the process has completed flushing its state.

It is possible for the reconfigurable process to, in effect, peek inside the $Reconfig$ stream to see if a reconfiguration had been requested, allowing the reconfigurable process to flush its state before the actual read of the $Reconfig$ stream. This is possible because typical hardware implementations of FIFOs provide as outputs status signals regarding the existence of data in the queue as well as the actual data value at the top of the queue. The set wrapper can therefore look ahead to determine if a reconfiguration event is at the top of the queue and, if required, flush its pipeline state before acknowledging the reconfiguration request by reading from the queue.

The synchronization scheme with the pipeline flushing enhancements can also be formalized using the CSP language. Communication in CSP occurs through point-to-point, unidirectional channels. Figure 3.10 shows two processes, O and P, connected by a single channel, c. Assuming that O does nothing but write to the channel and process P does nothing but read from the channel, the processes can be expressed as

$$O = c!x \rightarrow O$$

$$P = c?x \rightarrow P$$

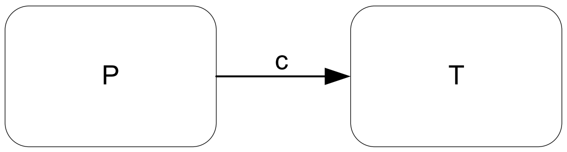


Figure 3.10: Two processes sharing single channel.

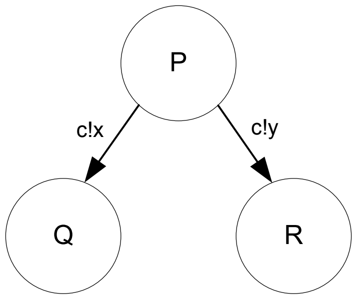


Figure 3.11: Process transition diagram.

with ! indicating transmission of x and ? indicating reception of x.

Development environments using the CSP model of computation do not permit processes to be swapped out during run-time. CSP, however, does permit a process to change its behavior as a result of communication with its environment using the choice operator, $|$. For example, a process, P, that behaves like process Q after receiving x from channel c or behaves like process R after receiving y is given by

$$P = (c?x \rightarrow Q) \mid (c?y \rightarrow R)$$

A graphical representation of this can be seen in Figure 3.11.

To describe the reconfiguration synchronization scheme in terms of CSP, each set of reconfigurable processes is treated as a single process that can take on the behavior of any process in the set in response to communication from the configuration controller. The processes within the set behave as normal, with the exception that after every N input samples the process polls the reconfiguration stream to determine if a reconfiguration is requested.

The equations that follow describe the system presented in Figure 3.12. The reconfigurable set, SET, has two channels, *set_in* and *set_out*, for processing data and one channel, *Reconfig*, for interfacing with the configuration controller $CONTROL$. SET implements P or Q when 0 or 1, respectively, is received on channel *Reconfig*.

$$SET = (Reconfig?0 \rightarrow P) \mid (Reconfig?1 \rightarrow Q) \tag{3.1}$$

Either process will return to implementing SET after N input samples have been processed so that SET can communicate with $CONTROL$.

$$\begin{aligned} P =& set\_in?x_1 \rightarrow set\_out!f_P(x_1) \rightarrow set\_in?x_2 \rightarrow set\_out!f_P(x_2) \rightarrow ... \\ & \rightarrow set\_in?x_N \rightarrow set\_out!f_P(x_N) \rightarrow SET \end{aligned} \tag{3.2}$$

$$\begin{aligned} Q =& set\_in?x_1 \rightarrow set\_out!f_Q(x_1) \rightarrow set\_in?x_2 \rightarrow set\_out!f_Q(x_2) \rightarrow ... \\ & \rightarrow set\_in?x_N \rightarrow set\_out!f_Q(x_N) \rightarrow SET \end{aligned} \tag{3.3}$$

$CONTROL$ also processes inputs on its channel *control_in*. After collecting M samples, $CONTROL$ makes a decision as to which reconfigurable process should be running, communicating this decision to SET via channel *Reconfig*.

$$\begin{aligned} CONTROL =& control\_in?x_1 \rightarrow ... \rightarrow control\_in?x_M \\ & \rightarrow Reconfig!f(x_1, ..., x_M) \rightarrow CONTROL \end{aligned} \tag{3.4}$$

While not a required component of a system, $FORK$ has been defined to formalize the

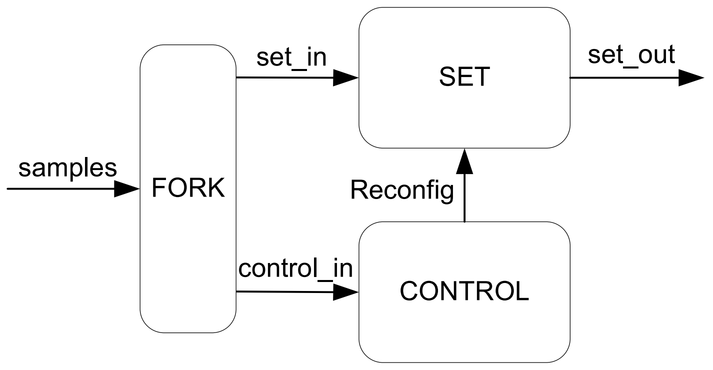


Figure 3.12: CSP description of reconfiguration synchronization scheme.

case where SET and $CONTROL$ process their input channels at different rates. $FORK$ feeds every input it receives to SET and every L^{th} input to $CONTROL$.

$$\begin{aligned} FORK = & samples?x_1 \rightarrow set\_in!x_1 \rightarrow ... \\ & \rightarrow samples?x_L \rightarrow set\_in!x_L \rightarrow control\_in!x_L \rightarrow FORK \end{aligned} \tag{3.5}$$

In CSP, parallel execution of processes is indicated with the || operator. Processes executing in parallel synchronize to each other on events that are common to their alphabets. In $SYSTEM$ these common events are comprised of communication over the shared channels.

$$SYSTEM = (FORK \mid\mid SET \mid\mid CONTROL) \tag{3.6}$$

While the absence of deadlock was proven using SDF, another concern of multi-process systems is livelock. Livelock occurs when two processes continually react to one another, making no progress in computations but continuing to execute. A common analogy for livelock is two people heading towards each other in a hallway. If both continual move towards the same wall at the same time to avoid the other, neither will make any headway. Instead, both persons will be continually moving from one wall to the other.

In the case of $SYSTEM$ livelock is not possible as, with one exception, the processes do not react to each other. The sole exception concerns the response of SET to a reconfiguration command from $CONTROL$. Though SET does react to $CONTROL$ by swapping the executing process, $CONTROL$ is incapable of modifying its behavior in response to SET.

3.6 Use Model

Identification of the end user is an important step in the creation of any design tool. While many in the HLS community target software programmers or scientific computing users as end users, a lack of standards and tool immaturity generally prevent a novice in hardware design from developing useful hardware applications. Few standards exist among FPGA board vendors to support the seamless porting of applications, requiring the user to create the interface logic between an FPGA and its peripherals in an HDL. Though a few HPC platforms integrate an FPGA with a processor, allowing HLS tool vendors to provide a complete end-to-end flow for targeted system, these integrated systems are not suitable for embedded applications.

Furthermore, as discussed in Chapter 5 and in [48], while current HLL-to-gates tools do accept as input code written in a standard HLL, the process of writing code in C for execution on a processor is different from writing C to be synthesized to hardware. Specifically, while the ability to code in C instead of HDL significantly increases productivity, the designer must think in terms of the hardware to be generated.

The tools and techniques presented in this dissertation, while raising the level of abstraction of dynamic hardware design, are hindered by these limitations in the current generation of HLS tools. As such, the targeted end user in this work is a hardware designer.

While the developed tools specifically target dynamic hardware development, the environment can be used to support a variety of other use models.

- Multiprocessor System Design

  Single-chip multi-processing is an area currently receiving great interest from industry and academia. Multi-processor designs are becoming the norm in many domains, including desktop, server, and embedded computing. FPGAs are an excellent implementation platform for multi-processor system owing to their inherit flexibility and the maturity of the soft-core processors offered by the FPGA vendors.

  The Xilinx MicroBlaze soft-core processor is small enough that a dozen or more processors to be placed even on the smaller FPGAs [79]. In the standard tool flow, system creation is accomplished through the Xilinx Embedded Development Kit (EDK). From within a GUI the designer can add peripherals, customize the processor, and compile software. However, the EDKs flexibility can be a detriment to the rapid creation of multi-processor systems. Each processor in the system must manually be connected to the system, as well as any peripherals the processors require. Finally, the developer must decide on a communication scheme (shared memory, direct connection) and create a software API to support the selected scheme.

  In contrast to the many steps required to implement a multi-processor system in the Xilinx EDK, this dissertation's development flow inherently supports multi-processor creation from an Impulse C description. Each CSP process in the design can be flagged for software implementation, resulting in the automatic generation and programming of a multiprocessor system.

- Mixed-IP Design

  Given the NRE costs of IP development, companies attempt to reuse existing IP wherever possible. While HLL-to-gates tools facilitate the rapid creation of hardware IP, the performance is not on par with IP created by an experienced hardware designer [52]. For many applications the reduced time-to-market and lower development costs provided by these HLL-to-gates tools outweigh the performance penalties. However, if superior IP exists a method for integrating IP into the development flow is required.

The tools created for this dissertation greatly simplify the addition of existing IP. To facilitate high-level simulation and integration, the designer describes the existing IP in a HLL, possible porting existing simulation code. The high-level simulations will then correctly model the final system. After the frontend tool flow has been run the designer simply edits the RCSF file to point to the synthesized netlist of the existing IP. The implementation flow will then pull the existing IP into the system.

- Rapid Prototyping of Static Designs

  HLS tools, such as Impulse C, facilitate the rapid prototyping of static hardware in FPGAs. However, for common cases these tools do not automatically produce an implementable design. Impulse C, for example, creates a design description in synthesizable HDL, requiring the application developer to integrate this hardware with his or her system before implementing the designs outside of the Impulse C environment. For a few supported platforms HLS tools may go further and automate the integration of the generated hardware with the platform, though bitstream creation is still the user's concern. In contrast, this work's development flow automates all stages of system implementation, including system integration and bitstream creation.

Building flexibility into the development environment is key to incorporating multiple use models into a single environment. Stable and known interfaces into the different stages of the development flow enable a user to see the environment as an unconstrained, multi-purpose tool that facilitates design exploration. For this reason the implementation of this methodology utilized a modular approach with defined, human-readable exchange formats between components.

The methodology discussed in this chapter provides abstractions and techniques that facilitate the rapid creation of dynamically reconfigurable hardware. While this methodology is applicable to a variety of design entry environments, an implementation of these ideas using the Impulse C language allows the pros and cons to be quantified.

Chapter 4

Implementation

The previous chapter described the approach to simplifying dynamic hardware development. The actual implementation of this approach is the focus of this chapter. While the commercial HLS tools perform the generation of the HDL from an HLL, much work remains to be done to produce a deployable design. Tools created or modified for this dissertation enable high-level simulation of dynamic hardware, floorplan the design, create a clocking scheme, insert a configuration controller, generate partial bitstreams using Xilinx tools, and package the bitstreams in a simple file system for retrieval at run-time. The complete tool flow integrates more than a dozen tools into a single, automated flow.

This chapter begins with a discussion of the design capture methodology, including the language extensions defined in this dissertation to describe partial reconfiguration. The results of the tool flow are then described, followed by an overview of each tool and file format created for this dissertation.

4.1 Design Capture

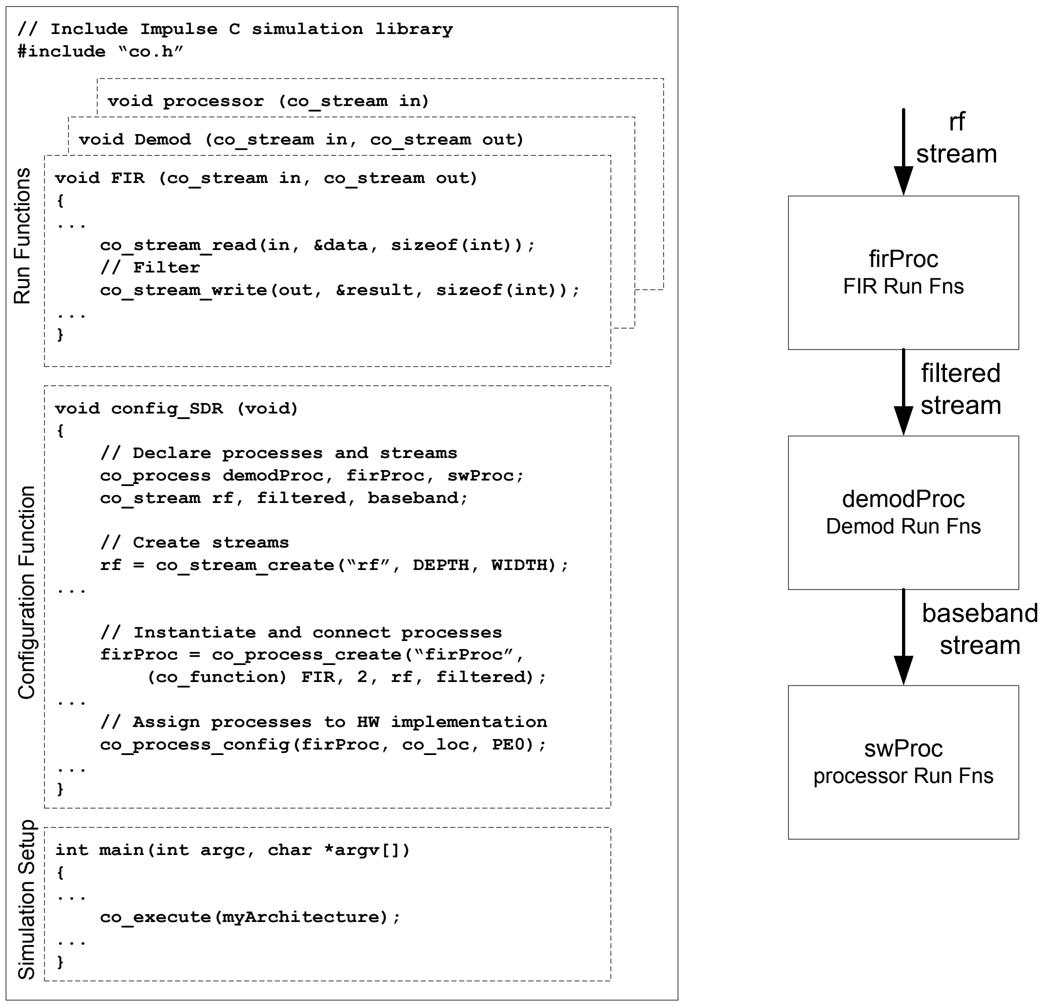
While many HLS development environments exist, Impulse C was selected as the design entry format owing to the suitability of its CSP-based model of computation and the availability of tools and source code.

Impulse C [48] is an ANSI C-based language utilizing the same stream and process abstractions as Los Alamos National Lab's Streams-C work [43]. Based on the CSP model, Impulse C permits the application developer to describe hardware using a large subset of ANSI C.

Figure 4.1 illustrates a simple SDR application described in Impulse C. The developer first decomposes the application into concurrently running processes connected by FIFO-based streams. A process is defined as a standard C function, called the process's *run function*, which accepts a special stream data type as arguments. Calls to the Impulse C functions `co_stream_read` and `co_stream_write` read from or write to a stream's FIFO buffer, respectively. These are blocking function calls, in sticking with the CSP communication model, with writes blocking until the destination is ready to accept data and reads blocking until data is available.

The connectivity between processes is specified by the application's *configuration function.* The configuration function also assigns processes to hardware or software implementation. As Impulse C is ANSI C compliant, a `main()` function exists. When compiled against the Impulse C simulation library, an executable is produced that simulates the application's behavior by spinning off each process's run function as a separate thread. When compiled to hardware using the CoDeveloper toolset, any process marked for hardware implementation is compiled to HDL and wrapped in a interface of the user's choosing.

While the Impulse C tools provide an excellent high-level development environment for FPGA applications, no provisions exist for describing dynamic hardware. Through the addition of new functions and slight modifications to the behavior of the existing tools, the



```
// Include Impulse C simulation library
#include "co.h"

void processor (co_stream in)
void Demod (co_stream in, co_stream out)
void FIR (co_stream in, co_stream out)
{
...
    co_stream_read(in, &data, sizeof(int));
    // Filter
    co_stream_write(out, &result, sizeof(int));
...
}

void config_SDR (void)
{
    // Declare processes and streams
    co_process demodProc, firProc, swProc;
    co_stream rf, filtered, baseband;

    // Create streams
    rf = co_stream_create("rf", DEPTH, WIDTH);
...

    // Instantiate and connect processes
    firProc = co_process_create("firProc",
        (co_function) FIR, 2, rf, filtered);
...
    // Assign processes to HW implementation
    co_process_config(firProc, co_loc, PE0);
...
}

int main(int argc, char *argv[])
{
...
    co_execute(myArchitecture);
...
}
```

Figure 4.1: Example Impulse C application.

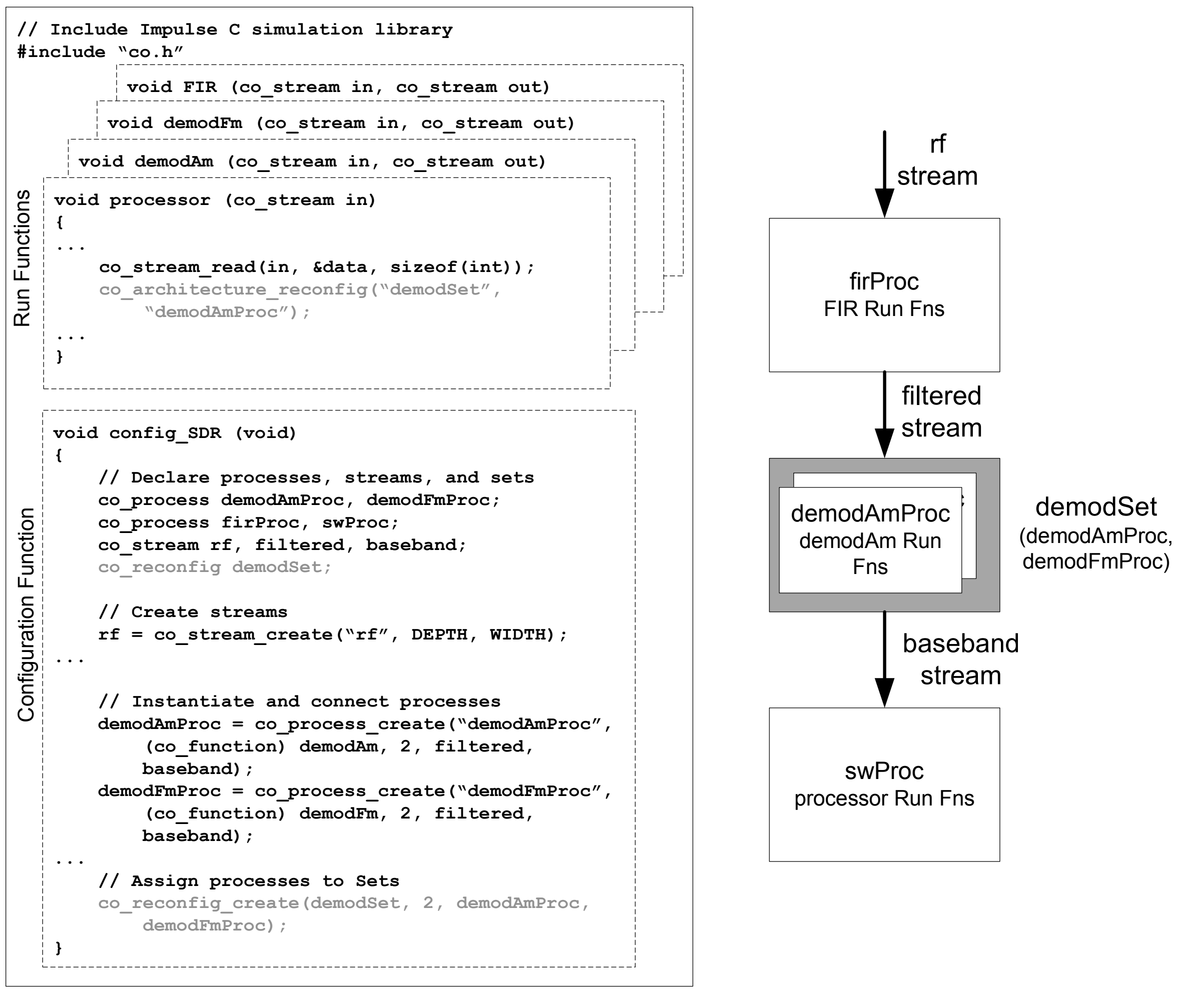
Impulse C language becomes a powerful development framework for dynamic reconfiguration of FPGA hardware.

Modifications to the CoDeveloper simulation library and corresponding extensions to Impulse C have been made permitting dynamic hardware to be simulated at a high level [1]. This modified language is referred to as DR Impulse C, highlighting its Dynamic Reconfiguration (DR) ability.

The Impulse C simulation library stores all information concerning the system's architecture in a data structure called `the_arch`. The modifications to the simulation library involve extending this data structure to include information about the reconfigurable processes. Impulse C simulations are multi-threaded in nature. Each concurrently-running CSP process occupies a separate thread. Communication occurs through writes and reads to shared circular buffers in memory. Semaphores are used to correctly synchronize the communicating threads. To describe dynamic hardware a method for stopping running threads was developed that involves status flags added to the data structure describing each process. When a process attempts communication via `co_stream_read` and `co_stream_write` functions, the modified functions first check the process's status flag to verify that the process is still active. If the process has been stopped, the modified code safely kills the thread.

To describe RTR applications in DR Impulse C, the programmer defines sets of mutually exclusive Impulse C processes. Figure 4.2 highlights the language extensions in the context of the SDR application. The demodulation scheme can be swapped at run-time, as directed by a software configuration manager process. New functions are utilized to create a set of reconfigurable processes (`co_reconfig_create`) and select a new dynamic process to execute in hardware (`co_architecture_config`). Chapter 5 demonstrates the use of these language extensions for a variety of applications.

[1]Through an agreement with Impulse Accelerated Technologies, Inc., the CoDeveloper Impulse C application development environment has been obtained, along with the source code to the Impulse C simulation library.



```
// Include Impulse C simulation library
#include “co.h”
        void FIR (co_stream in, co_stream out)
      void demodFm (co_stream in, co_stream out)
    void demodAm (co_stream in, co_stream out)
  void processor (co_stream in)
  {
  ...
      co_stream_read(in, &data, sizeof(int));
      co_architecture_reconfig(“demodSet”,
          “demodAmProc”);
  ...
  }

  void config_SDR (void)
  {
      // Declare processes, streams, and sets
      co_process demodAmProc, demodFmProc;
      co_process firProc, swProc;
      co_stream rf, filtered, baseband;
      co_reconfig demodSet;

      // Create streams
      rf = co_stream_create(“rf”, DEPTH, WIDTH);
  ...

      // Instantiate and connect processes
      demodAmProc = co_process_create(“demodAmProc”,
          (co_function) demodAm, 2, filtered,
          baseband);
      demodFmProc = co_process_create(“demodFmProc”,
          (co_function) demodFm, 2, filtered,
          baseband);
  ...
      // Assign processes to Sets
      co_reconfig_create(demodSet, 2, demodAmProc,
          demodFmProc);
  }
```

Figure 4.2: Example DR Impulse C application.

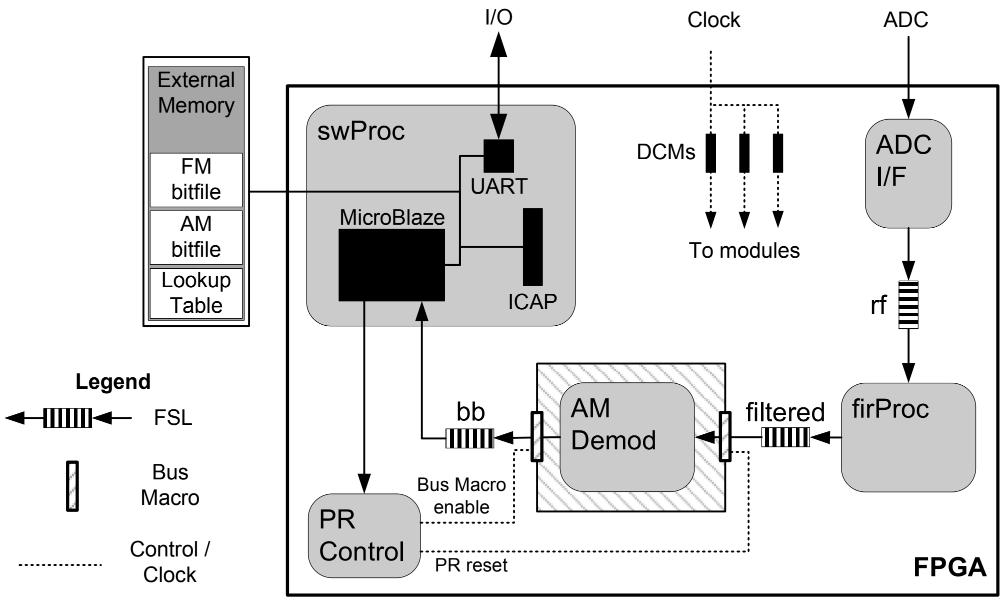


Figure 4.3: Final design implementation.

4.2 Produced Design

Working solely from the application designer's DR Impulse code and a Board Support Package, the automated flow produces a bitstream ready for FPGA programming, along with a memory image containing the partial bitstreams and the scripts required to load both onto the target platform.

Figure 4.3 presents an example implementation, based on the SDR application presented in the previous section. In this example an Analog-to-Digital Converter (ADC) supplies data samples to an FIR filter, which in turn feeds a demodulator module. The demodulator module may be dynamically reconfigured to support either AM or FM waveforms. All data in the design are communicated using the FIFO-based Fast Simplex Link (FSL), supporting asynchronous reads and writes.

Several important aspects of this dissertation are presented in Figure 4.3. The Partially Reconfigurable (PR) module in the figure has been area constrained to a specific location of the FPGA by the Floorplanner tool. All non-reconfigurable modules are unconstrained, permitting the Xilinx tools to choose their optimum locations.

All non-clock signals crossing the boundary between the static and PR regions must pass through a bus macro. During the actual reconfiguration process the logic internal to the region undergoing PR is in an undefined state. To stop the internal logic from producing random outputs that affect the reset of the system, the bus macro on the output of a PR region can be disabled. The tool flow automatically creates a PR Control module for each PR region that disables the bus macros before reconfiguration and places any newly reconfigured module into a known, good state by toggling the module's reset line.

Control of partial reconfiguration is handled by a MicroBlazed-based system running the user's control code. When a `co_architecture_reconfig` instruction is encountered the MicroBlaze reads a Look-Up Table (LUT) from an external storage device. This LUT contains a list of all partial bitstreams and their locations in external storage. Once the correct bitstream has been found, the MicroBlaze disables the bus macros on the PR module and streams the partial bitstream to the ICAP. When complete the PR module is reset and the bus macros enabled.

CSP permits each process to run at its own speed. To replicate this in hardware, each process receives its own clock, subject to resource availability. The FSL connections between processes are implemented as asynchronous FIFOs to enable cross-clock domain communication. The clocking structure is automatically generated using timing estimates from the synthesis tool.

4.3 Tools

The complete tool flow is tied together using Make, a dependency tracking utility commonly used in software development projects to automate the building of applications [80]. Commands for generating each file, as well as the dependencies between files, are expressed in a Makefile that is passed to the Make utility. By only regenerating files whose dependencies

```
HEADER_FILE = img.h
HARDWARE_FILE = img_hw.c
SOFTWARE_FILE = img_sw.c
ARCH_NAME = img
BSP = ../bsps/memec/memec.xml
```

Figure 4.4: Project Makefile.

have changed since the last build, Make speeds the building process. Though not shown in Figure 4.5 or Figure 4.10, Make encompasses both the frontend and backend flows, from HLL simulation to FPGA programming.

The designer edits a local project makefile shown in Figure 4.4, providing a link to the input files and specifying the target architecture's Board Support Package (BSP) for implementation. This local project makefile is read by the main Makefile. These are the only settings a user need provide the flow.

4.3.1 Frontend

As discussed above, Impulse C was selected as the design capture language. The CoDeveloper toolset compiles Impulse C to synthesizable HDL. Extensions to Impulse C created for this dissertation permit Dynamically Reconfigurable (DR) hardware to be described from an HLL. This modified language is called DR Impulse C.

The frontend flow, shown in detail in Figure 4.5, consists of the CoDeveloper Toolset for generating HDL from an HLL, the Preprocess script for creating the Reconfigurable Computing Specification Format (RCSF) file, and the GCC compiler for creating a simulation executable.

The Preprocess tool prepares the DR Impulse C code for compilation by the CoDeveloper tools. The extensions to Impulse C implemented in this dissertation are not recognized by the CoDeveloper tools. As shown in Figure 4.6, the Preprocess script parses the designers DR Impulse C code, converting it into the generic Impulse C code that the CoDeveloper tools

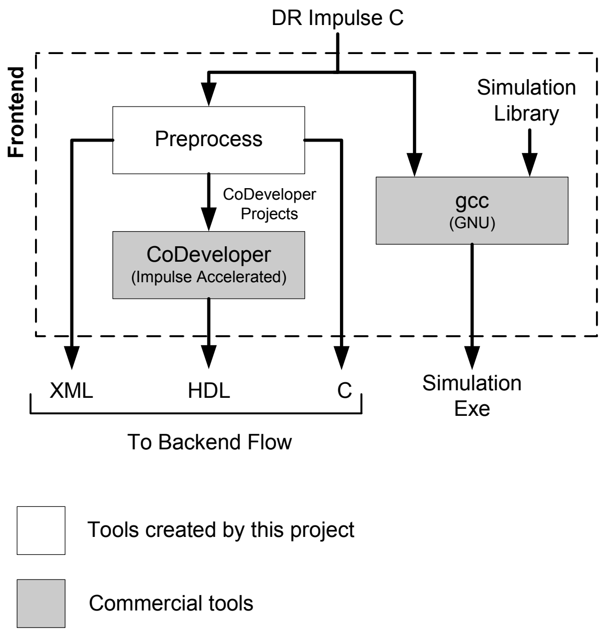


Figure 4.5: Frontend Tool Flow.

understand by dividing the DR Impulse C project into a series of Impulse C projects, one for each CSP process in the design. Each CSP hardware process is then taken through the CoDeveloper compilation step individually. Software processes in the project are converted to ANSI C suitable for compilation for the target processor. This modularization of the design facilitates IP reuse and design modifications further downstream in the flow.

The Preprocess script also generates an XML description of the design in the RCSF. This XML description, discussed in the next section, is human-readable and contains only the essential, architecture-agnostic information for the design. By editing this file additional IP can be added to the design, permitting different component implementations to be substituted according to the specific implementation requirements.

4.3.2 Design Exchange

To permit seamless porting of applications between platforms, the architecture-agnostic frontend flow is decoupled from the architecture-specific backend flow. The backend flow accepts

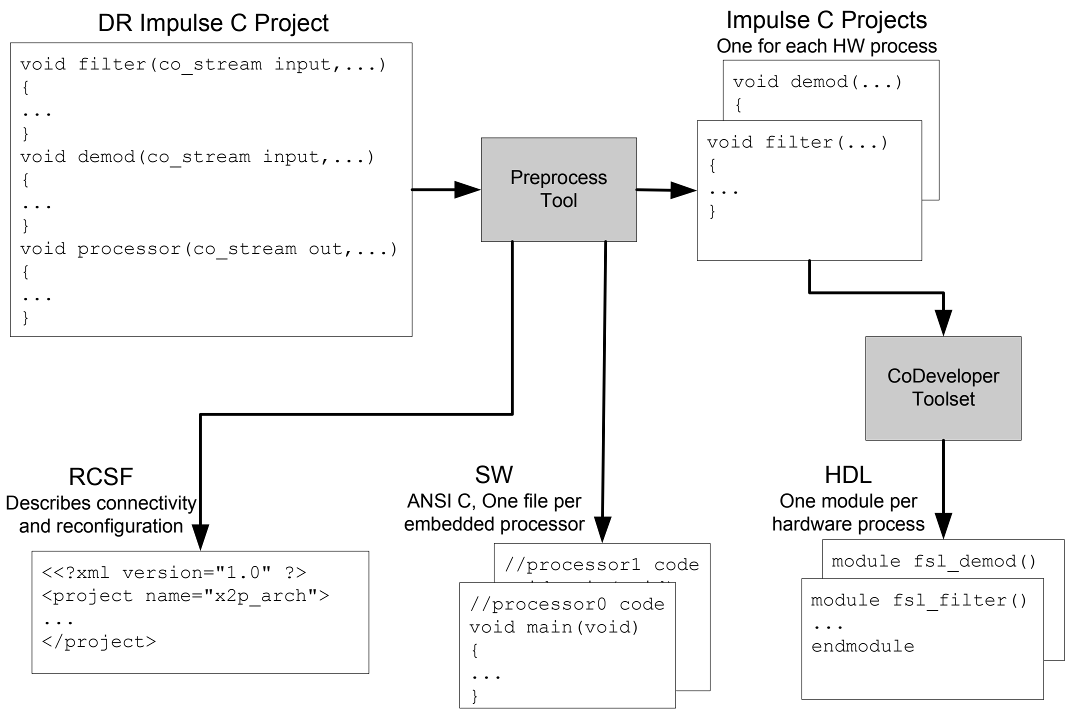


Figure 4.6: Preprocess usage.

the outputs of the frontend flow (HDL, C code, and RCSF) as well as a BSP describing the target platform.

This dissertation utilizes XML as the mechanism for capturing architecture-agnostic information about a design in the Reconfigurable Computing Specification Format (RCSF). The XML listing shown in Figure 4.7 presents the retargetable design specification for a simple application.

Each process in the design has a separate entry capturing the I/O streams, location of the process's HDL, and the type of process: dynamic, static, software, or IP. Software processes are mapped to an embedded soft processor. IP processes use existing hardware that is included with the BSP. All I/O to the FPGA is controlled by software or IP processes. Each dynamic process in a reconfigurable set must share the same module name.

Each set of reconfigurable processes has its own entry in the RCSF indicating which processes are in the set, as well as which process is the default in hardware. As the automated

```
<?xml version="1.0" ?>
<project name="CaseChange">
    <process folder="hw_upper" module_name="fsl_SetA" name="upper" type="dynamic">
        <stream direction="input" name="s2" port_name="FSL0_S"/>
        <stream direction="output" name="s1" port_name="FSL1_M"/>
    </process>
   ...
    <process co_arch_reconfig="true" folder="sw_process0" name="proc0" type="sw">
        <stream direction="input" name="s1" port_name="FSL0_S"/>
        <stream direction="output" name="s2" port_name="FSL1_M"/>
    </process>
    <reconfig index="0" name="SetA">
        <process_name default="true" index="0" name="upper"/>
        <process_name index="1" name="lower"/>
    </reconfig>
</project>
```

Figure 4.7: Retargetable Design Specification.

implementation flow tracks reconfigurable processes based on numbers, instead of names, an index number is provided linking the process's name with its number in the implementation flow.

The Board Support Package (BSP) contains all information about a platform required by the implementation flow, including peripherals, implementation constraints, and compilation libraries. This information is organized into the directory structure in Figure 4.8.

All peripherals and soft processors are stored as Xilinx-proprietary NGC netlists, instead of HDL, to simplify and speed implementation. For every peripheral supported by the BSP, there is a corresponding netlist in the `netlist` directory and, where required, a UCF constraints file, a BMM memory mapping file, and an ELF executable in `data`. Soft processors add a script to compile code and a directory containing the compilation libraries in the `src` directory. Platform-specific scripts to download bitstreams and program the FPGA's external memory with the partial bitstreams are found in the `etc` directory.

```
BSP_NAME/
       BSP_NAME.xml
       data/
              BSP_NAME.ucf PROC0_NAME.ucf PROC0_ports.vhd PROC0.bmm
       etc/
              download.sh xmd.scr PROC0_elf.sh
       netlist/
              PROC0.ngc PRR_controller.ngc fsl_stream.ngc
              busmacros/
                     busmacros.v busmacro_xc2vp_l2r_async_narrow.nmc ...
       src/
              PROC0_NAME/
                     reconfig.c xhwicap.h ...
```

Figure 4.8: BSP directory structure.

An XML file specifies all of the peripherals and soft processors contained in the BSP. This file, shown in 4.9, organizes all required information about the board, including number of internal clocks available, clock frequency, and device number. For each soft processor or peripheral this file specifies placement keep-out regions, clock requirements, and internal port names.

4.3.3 Backend

The architecture-specific implementation flow accepts the RCSF file, HDL modules, and C code from the frontend. In addition, a Board Support Package (BSP) must be specified, supplying all the platform-specific information required to produce a deployable design. The implementation tool flow, shown in Figure 4.10 integrates multiple applications, including tools automating placement, HDL generation, and clock creation. These tools are discussed in the following sections.

The tool flow shown above in Figure 4.10 is responsible for transitioning the user's design to a deployable format of an initial bitstream and a memory image containing the partial

```
<?xml version="1.0" ?>
<platform name="Memec" reset="active-low" device="xc4vlx25-ff668-10" clock="100MHz"
        clock_location="top" num_clocks="16">
        <description>Text description of platform and peripherals.</description>
        <processor name="Microblaze" compiler="mb-gcc" prefix="mb_cntl"
               max_instances="1" uart="true" icap="true" reset="active-low" ddr="true"
              clock="50MHz" dcm="true" keepout="0,16 - 2 ,64" num_clocks="5"/>
        <processor name="Microblaze Plain" compiler="mb-gcc" prefix="mb_plain"
               reset="active-low" dcm="false" max_instances="0" clock="100MHz"/>
        <peripheral name="LED Driver" reset="active-high" dcm="false"
               prefix="ledDriver" max_instances="1" clock="100MHz">
               <port direction="input" prefix="FSL0_S"/>
        </peripheral>
</platform>
```

Figure 4.9: Board Support Package Specification.

bitstreams. Like all aspects of the automated flow, backend implementation is controlled by Make. While Make is an excellent tool for managing the generation of files and executables, it is limited in its parsing and processing capabilities. Because of these limitations, it is the Postprocess tool that handles the bulk of the backend file generation.

The Postprocess script parses the XML RCSF file generated by the Preprocess tool and the BSP. During the course of implementation, Postprocess can act in three different modes:

1. XST: Postprocess creates XST scripts to automate the generation of a netlist from the CoDeveloper-created HDL.

2. HDL: After synthesis has occurred, Postprocess examines the synthesis logs to determine a module's resource requirements and estimated clock frequency. The clocking information is compared against the FPGA's available clocking resources to generate a set of clocks. The application's RCSF file is parsed and any software processes are mapped to soft processors available on that board. The output of this mode is a top-level Verilog netlist, instantiating each process in the design, and a settings file for the

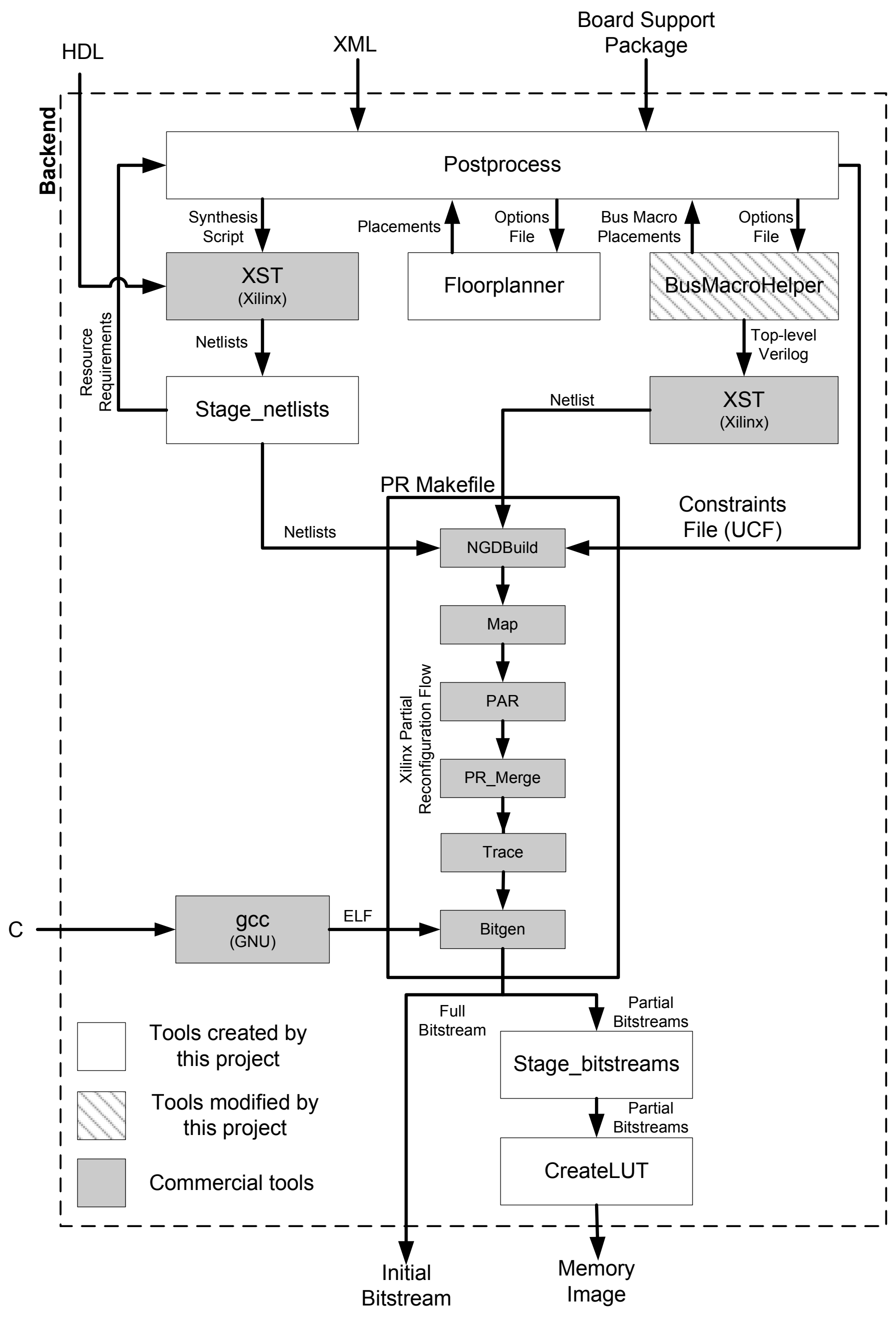


Figure 4.10: Backend Tool Flow.

Floorplanner tool.

3. BMH: Postprocess parses the output of Floorplanner to create a set of UCF constraint files and a settings file for the BusMacroHelper (BMH) tool. This mode produces all of the constraint files required for implementation by the Xilinx tools.

The Stage_netlists script copies the synthesized netlists for each CSP process into the appropriate directories in the implementation flow directory structure.

The Floorplanner utility is responsible for creating area constraints for each reconfigurable region of the FPGA. This tool accepts as input as list of the resource requirements of each set and a list of keep-out regions. The keep-out regions correspond to areas of the FPGA that must be available for peripherals or soft processors, such as regions near critical I/Os. In keeping with other FPGA floorplanning projects [81] [82] [83], Floorplanner uses a Simulated Annealing (SA) algorithm to find a near-optimal minimum of a cost function. While other techniques, such as Integer Linear Programming, can guarantee the solution is optimal, the computational requirements can be higher than SA.

Unlike most previous work, Floorplanner has knowledge of the device's configuration architecture and attempts to find placements that minimize reconfiguration overhead. For the Virtex-II and Virtex-II Pro architectures, where configuration frames run the entire height of the device, this involves finding a solution that has a high aspect ratio (height versus width) to use as much of the configuration frame as possible for the reconfigurable module. In the Virtex-4 architectures, where configuration frames are 16 CLBs tall, Floorplanner places all modules on configuration frame edges.

Floorplanner starts by first populating a list of possible module placements, called realizations. All possible module placements are considered in the creation of this realization list, with placements that are overly wasteful of resources being removed. Once a list of acceptable placements has been created, SA is performed to minimum the cost function

$$cost = 10,000 * overlap + 10 * aspectError + waste + distance$$

Module overlap, contained in *overlap* as the sum of all overlapping CLBs, is weighted orders of magnitude higher in the cost function to ensure that no two PR regions will overlap. *aspectError* penalizes the placements for having a poor aspect ratio while *waste* is a measure of extra resources within the placement that will not be utilized on the device. The *distance* variable represents the total distance between reconfigurable regions and is used to minimize routing delays between reconfigurable regions.

Floorplanner is ignorant of static modules in the design beyond any keep-out regions that are specified in the BSP. This is not seen as a major limitation as the Xilinx placement tools are superior at static placement. The addition of static placement into Floorplanner would also require consideration of connectivity in the design and, for optimal results, some notion of timing. Figure 4.11 shows a sample placement performed by Floorplanner. Upon selection of a suitable placement, the tool produces the textual display of the solution shown in the figure.

To produce the partial bitstreams requires a special path to the standard Xilinx ISE toolset. These modified tools make up the Xilinx Early Access PR (EAPR) Flow. Included are tools to synthesize HDL into netlists (XST), translate netlists into Xilinx-standard databases (NGDBuild), place-and-route the design (PAR), analyze timing (TRCE) and generate a bitstream (Bitgen).

The Xilinx EAPR Flow requires that special connection points, called bus macros, surround reconfigurable modules. These bus macros are fixed in location providing stable connection points for the reconfigurable modules. BusMacroHelper is a tool developed for a related project that was modified for this dissertation to correctly connect enable signals to bus macros. These enable signals are required to disable a module's outputs during reconfiguration. BusMacroHelper accepts the top-level Verilog netlist created by the Postprocess script and a list of reconfigurable modules. The tool breaks the connections to these modules, inserting bus macros between the static and reconfiguration regions.

The Stage_bitstreams script copies the partial bitstreams created by the automated imple-

```
@ RECONFIG_SET0: (19, 51) -- (20, 8)
# RECONFIG_SET1: (34, 31) -- (42, 8)
* RECONFIG_SET2: (11, 31) -- (14, 8)
^ RECONFIG_SET3: (25, 27) -- (27, 4)

% dram_keepout:   (0, 64) -- (2, 16)
O V2P_PPC:        (7, 47) -- (14, 32)
/ V2P_PPC:        (31, 47) -- (38, 32)

I  B      B      B      B      B      B      B      B  I
I  B      B      B      B      B      B      B      B  I
I  B      B      B      B      B      B      B      B  I
I  B      B      B      B      B      B      B      B  I
I%%B%     B      B      B      B      B      B      B  I
I%%B%     B      B      B      B      B      B      B  I
I%%B%     B      B      B      B      B      B      B  I
I%%B%     B      B     @B@     B      B      B      B  I
I%%B%    OBOOOOOOBO    @B@     B     /B//////B/      B  I
I%%B%    OBOOOOOOBO    @B@     B     /B//////B/      B  I
I%%B%    OBOOOOOOBO    @B@     B     /B//////B/      B  I
I%%B%    OBOOOOOOBO    @B@     B     /B//////B/      B  I
I%%B%     B   ***B*    @B@     B      B  ####B##### B  I
I%%B%     B   ***B*    @B@    ^B^^    B  ####B##### B  I
I%%B%     B   ***B*    @B@    ^B^^    B  ####B##### B  I
I%%B%     B   ***B*    @B@    ^B^^    B  ####B##### B  I
I  B      B   ***B*    @B@    ^B^^    B  ####B##### B  I
I  B      B   ***B*    @B@    ^B^^    B  ####B##### B  I
I  B      B      B      B     ^B^^    B      B      B  I
I  B      B      B      B      B      B      B      B  I

LEGEND:   B = BRAM  I = IOB
```

Figure 4.11: Example Floorplanner placement for four PR regions on xc2vp30.

mentation flow to the bitstreams directory. Stage_bitstreams renames the partial bitstreams to a human-readable format.

The CreateLUT tool performs two tasks. It creates a binary Look-Up Table (LUT) that lists the size and location in memory of each partial bitstream. This LUT is used by the MicroBlaze configuration controller to find the desired partial bitstream in memory. Secondly, the script concatenates the LUT and the partial bitstreams together into a single memory image, simplifying application execution.

The automated tool flow described in this chapter abstracts low-level details away from the application designer while eliminating the many manual, error-prone tasks that plague dynamic hardware development. These tools enable the creation of a diverse set of applications commonly encountered by FPGA designers.

Chapter 5

Application Development

To demonstrate the capabilities of this dissertation's development environment, designs were implemented from a broad sampling of streaming applications, including cryptography, SDR, and video processing. In some of these applications all hardware was generated from a high-level description, while in others existing IP was utilized. Several of these applications were also implemented manually, using the existing Xilinx implementation flow, to provide comparison points from which to judge the productivity enhancements and performance penalties of this high level development environment.

This first section steps through a video processing application, detailing the application development process in this high level development environment. Additional properties of the development environment are highlighted by applications described in the following sections.

5.1 Video Processing

A video processing application has been implemented using this development flow with a camera's video stream filtered in real time with one of several filters and displayed on a monitor. A separate filter acts on each of three colors and each can be independently reconfigured to implement an edge detector, a median image filter, or a pass-through. Video processing is considered representative of streaming applications that can benefit from reconfigurable hardware. The steps required to implement this application are described below.

5.1.1 Application Description and Simulation

In this application all reconfigurable hardware and configuration control logic are described in DR Impulse C. However, while no HDL need be written to create an application in this high-level development environment, hardware design knowledge is required. The first step in the creation of an application is partitioning. Use of the CSP model requires that operations to be implemented in reconfigurable hardware be divided into separate processes.

For many designs this is a straightforward procedure. Distinct operations are assigned to independent processes. The user should be cognizant of the limitations in HLS tools. The CoDeveloper toolset performs synthesis on each process independently of the others. Grouping only related functionality in a process can improve results from the tools. If existing IP is available for some or all of the application, the partitioning can be performed around the available hardware modules.

In the case of the video processing application, the video enters the FPGA as a continuous stream of pixels, row by row. The implemented filters (edge detection, smoothing, and median filtering) operate on a square window of pixels. Several methods exist for converting a stream of pixels into a window. These different methods may produce different CSP topologies. The designer must evaluate these methods to determine which is best given the hardware constraints of the application.

For example, a frame buffer could be constructed from external memory, as internal resources are insufficient for buffering a full frame, with one process pushing the video stream to external memory and another process reading out the desired window of pixels. While perfectly capable of meeting the area and through-put constraints, this approach requires

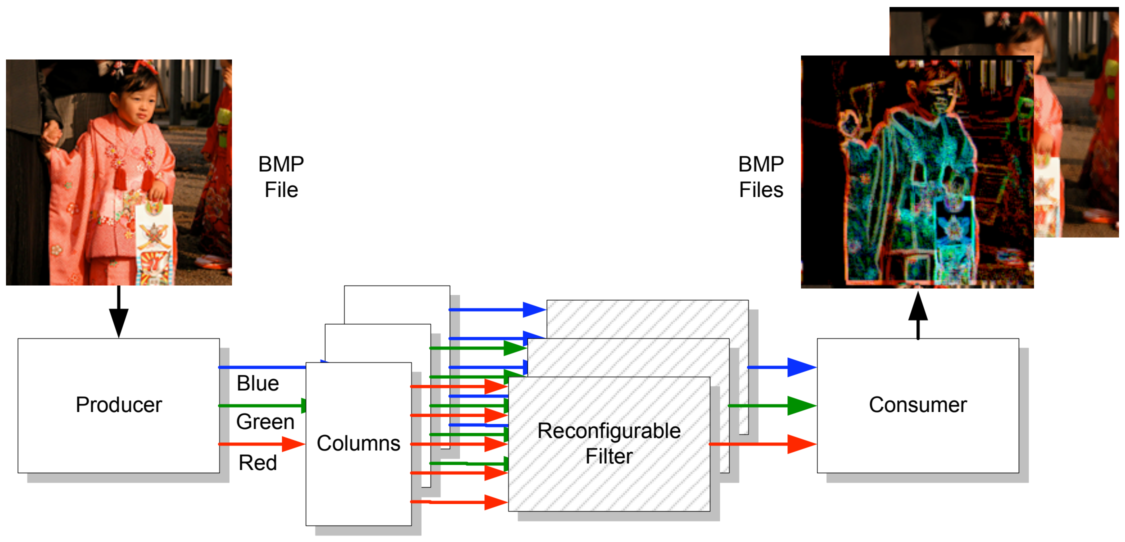


Figure 5.1: Video processing application.

the addition of a memory controller and the creation of FSL-to-memory interfaces. As the window size for effective edge detectors and median filters is not large, it was decided to implement buffers on the FPGA to create the window by storing five rows of pixels. This approach significantly reduced the complexity of the design.

Three separate filtering pipelines were constructed, one for each color. While a single pipeline was possible, separation of the colors increases the flexibility of the application, as each color's filter can be independently configured. Furthermore, as the filtering of one color shares no resources or data with another color, separation of the filters assists the HLS tools.

The final partition, shown in Figure 5.1, consists of two testbench processes, *Producer* and *Consumer*, three video stream buffers, *Columns*, and three sets of reconfigurable processes, each set containing identical filters. The Columns process buffers five rows of pixels, presenting a column of five pixels to the filters, which in turn buffer five of these columns to obtain a 5x5 window of pixels.

While HLS tools permit hardware to be generated from an HLL, the designer must be cognizant of tool and hardware limitations. An example of this is in the design of the median image filter. Median filters examine a windows of pixels around the target pixel, replacing the target pixel with the median value of its surroundings. Calculation of the median value of the window requires sorting to be performed.

The key requirement for this filter was the ability to process data at a rate matching the video capture card output of 27 MB/sec. Typical software approaches to finding the median value, using a sorting algorithm or a histogram, are not conducive to fast hardware implementation and are incapable of meeting the performance requirement without consuming significant FPGA resources.

A reasonable hardware design approach is to use a set of comparators to determine relations between the pixels in the window. Figure 5.2 shows an implementation of a 4-point median filter, replacing the target pixel with the median of the pixels above and below and left and right. Larger filters are possible, though the filter's area grows exponentially with the window size. The code can be pipelined by the HLS tool, producing a result every cycle. In contrast to a sorting algorithm, the latency of this approach is deterministic and the code can be easily pipelined. Note that, while the code is written in C, the approach is very similar to a design created in an HDL.

The DR Impulse C configuration function for this application, specifying the connections between processes, is shown in Figure 5.3. The three reconfigurable sets are created using the `co_reconfig_create()` function. Every filter instance in a set shares the same I/O connections, as is evident by the arguments to the `co_process_create()`. Processes that belong to a reconfigurable set are automatically targeted for hardware implementation.

The reconfiguration control code, shown in Figure 5.4, reconfigures the three reconfigurable sets serially, one after the other. Reconfiguration control is performed by a single process using of the `co_architecture_reconfig()` function.

The filters, configuration control logic, and testbenches are all described in DR Impulse C. For high-level simulation the two testbench processes load an input image from a Windows Bitmap (BMP) file, stream the pixels row by row to the filtering pipelines, and translate the filters' outputs to a BMP, as shown in Figure 5.1.

```
void medianFilterCross(co_stream r0, co_stream r1, co_stream r2,
    co_stream r3, co_stream r4, co_stream output_stream)
{
    co_uint8 a, b, c, d, pixelMag;
        ⋮
        a_gt_b = a > b; a_gt_c = a > c; a_gt_d = a > d;
        b_gt_a = b > a; b_gt_c = b > c; b_gt_d = b > d;
        ⋮
        if(a_gt_b & a_gt_c & a_gt_d)  // Determine largest value
            aGreatest = 1;
        else if(b_gt_a & b_gt_c & b_gt_d)
            bGreatest = 1;
        ⋮
        if(!a_gt_b & !a_gt_c & !a_gt_d)  // Determine smallest
            aLeast = 1;
        else if(!b_gt_a & !b_gt_c & !b_gt_d)
            bLeast = 1;
        ⋮
        if(aGreatest & bLeast)
            pixelMag = (c + d) >> 1;   // pixelMag is filter's output
        else if(aGreatest & cLeast)
            pixelMag = (b + d) >> 1; // Average 2 median pixels
        ⋮
}
```

Figure 5.2: Impulse C description of median filter.

```
void config_video(void *arg)
{
    // Declare Streams
    co_stream blue_source_pixeldata, green_source_pixeldata, red_source_pixeldata;
    co_stream blueColumn0, blueColumn1, blueColumn2, blueColumn3, blueColumn4;
        ⋮
    // Declare Processes
    co_process producerProcess, consumerProcess, controlProcess;
    co_process blueFilter, greenFilter, redFilter;
    co_process blueColumns, greenColumns, redColumns;
        ⋮
    // Declare Reconfigurable Sets
    co_reconfig BlueFilterSet, GreenFilterSet, RedFilterSet;
        ⋮
    // Create Streams
    blue_source_pixeldata = co_stream_create("blue_source", UINT_TYPE(12), 2);
    blueColumn0 = co_stream_create("blueColumn0", UINT_TYPE(12), 2);
        ⋮
    // Create and connect processes
    // PROCESS_INST = co_process_create("NAME", IMPULSE C FNS, # CONNECTIONS,
    //     CONNECTION LIST);
    blueColumns = co_process_create("blueColumns", (co_function) columns,
        6, blue_source_pixeldata, blueColumn0, blueColumn1, blueColumn3,
        blueColumn4);

    blueFilter = co_process_create("blueFilter", (co_function) medianFilterCross,
        6, blueColumn0, blueColumn1, blueColumn2, blueColumn3, blueColumn4,
        blue_result_pixeldata);

    blueEdge = co_process_create("blueEdge", (co_function) edge_detect,
        6, blueColumn0, blueColumn1, blueColumn2, blueColumn3, blueColumn4,
        blue_result_pixeldata);
        ⋮
    // Create and populate reconfigurable sets
    BlueFilterSet = co_reconfig_create("BlueFilterSet", 3, blueFilter,
        bluePassthru, blueEdge);
        ⋮
}
```

Figure 5.3: DR Impulse C configuration function for video filter.

```
void controller(void)
{
    volatile int delay;

    while(1)
    {
        // Delay
        for(delay = 0; delay < 40000000; delay++)
        {}

        printf("Reconfiguring for pass thru!\r\n");
        co_architecture_reconfig("BlueFilterSet","bluePassthru");
        co_architecture_reconfig("GreenFilterSet","greenPassthru");
        co_architecture_reconfig("RedFilterSet","redPassthru");
        ⋮
    }
}
```

Figure 5.4: DR Impulse C configuration control function for video filter.

5.1.2 Implementation

Implementation begins with the selection of a target platform. The development environment currently contains three BSPs targeting the Xilinx University Program (XUP) Virtex-II Pro development board, Memec Design's Virtex-4 MB development board, and Harris's Morpheus SDR platform. The only platform supporting video I/O is the XUP. Selection of the XUP as the target is made by editing the project's make file (Figure 4.4) to point to the proper BSP.

The DR Impulse C application used for high-level simulation includes two testbench processes that are not suitable for implementation. The BSP for the target FPGA platform contains an IP block for interfacing with both an attached video capture card and the on-board video DAC. The application's automatically created RCSF file, shown in Figure 5.5 includes the two testbench processes, Producer and Consumer. To interface with the actual hardware on the target platform, the RCSF must be edited to replace these processes with

```
<?xml version="1.0" ?>
<project implementation="xilinx_fsl" name="img_arch">
    <!-- Software testbenches for high-level simulation -->
    <process co_arch_reconfig="true" folder="sw_producerProcess"
        name="producerProcess" type="sw">
        <stream direction="output" name="blue_source_pixeldata" port_name="FSL0_M"/>
        <stream direction="output" name="green_source_pixeldata" port_name="FSL1_M"/>
        <stream direction="output" name="red_source_pixeldata" port_name="FSL2_M"/>
    </process>
    <process folder="sw_consumerProcess" name="consumerProcess" type="sw">
        <stream direction="input" name="blue_result_pixeldata" port_name="FSL0_S"/>
        <stream direction="input" name="green_result_pixeldata" port_name="FSL1_S"/>
        <stream direction="input" name="red_result_pixeldata" port_name="FSL2_S"/>
    </process>
        ⋮
</project>
```

Figure 5.5: RCSF file for simulated video processing design.

IP from the BSP for interfacing with the platform's video I/O. As seen is Figure 5.6, this edit involves the modification of only ten lines of XML code.

With modifications to the RCSF XML complete, the entire implementation process is performed with a single command, `make bits`. If desired for debugging purposes a static, non-RTR version of the design can be created with `make static`. In either case commands are provided to automate deployment to the target platform. The command `make download` places an initial bitstream on the board and then connects to the configuration controller to download an external memory image containing the partial bitstreams and the look-up table required for the configuration controller to find the correct bitstream.

The runtime of the tool flow depends heavily on the number of reconfigurable sets and the number of modules in each set. For the video processing application, with three reconfigurable sets of three modules each, nearly an hour was required to run through the implementation flow as the Xilinx EAPR flow must run the implementation flow once for each module that can be dynamically reconfigured.

```
<?xml version="1.0" ?>
<project implementation="xilinx_fsl" name="img_arch">
    <!-- IP video interface from BSP -->
    <process module_name="vid_capt" name="VC0" type="ip">
        <stream direction="output" port_name="MFSL_R" name="red_source_pixeldata"/>
        <stream direction="output" port_name="MFSL_G" name="green_source_pixeldata"/>
        <stream direction="output" port_name="MFSL_B" name="blue_source_pixeldata"/>
        <stream direction="input" port_name="SFSL_R" name="red_result_pixeldata"/>
        <stream direction="input" port_name="SFSL_G" name="green_result_pixeldata"/>
        <stream direction="input" port_name="SFSL_B" name="blue_result_pixeldata"/>
    </process>
        ⋮
</project>
```

Figure 5.6: RCSF file for implemented video processing design.

The implemented design, the layout of which is seen in Figure 5.7, encompasses 63% of a Xilinx Virtex-II Pro-30 FPGA. The filters operate at 57 MHz, sufficiently fast to support the incoming 640x480 video stream at 60 Hz. The total area of the all reconfigurable filters is 1,707 slices while the floorplanned reconfigurable regions consume 1,328 slices, resulting in an area savings of 379 slices from using PR. Any additional filters added to the system would increase this area savings.

5.2 Software Defined Radio

The ideal SDR is a single processor connected to an Analog-to-Digital Converter (ADC), digitizing signals directly from an antenna. Computational requirements, however, generally prohibit this as even a basic band-pass filter can require billions of multiply-accumulate operations per second. These demands necessitate the inclusion of hardware accelerators into an SDR.

An AM radio receiver was created targeting Harris's Morpheus SDR platform. The Morpheus includes four Xilinx Virtex 4 LX-60 FPGAs along with an ARM processor, all on a

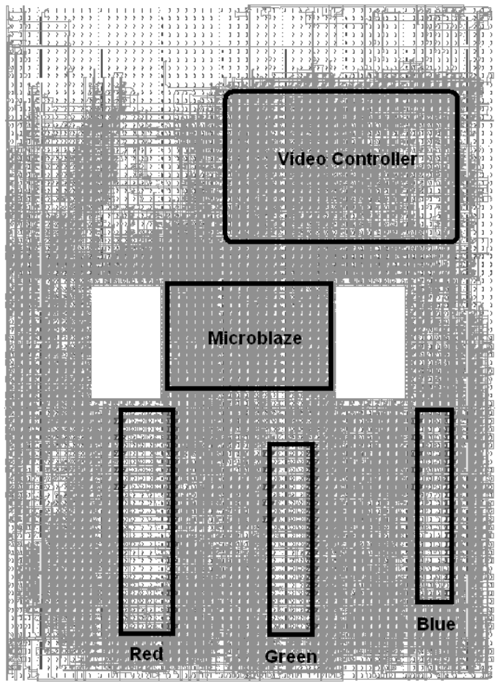


Figure 5.7: Floorplan of implementation in a Xilinx XC2VP30.

System-In-Package (SIP). The Morpheus development board contains the SIP along with the RF electronics and ADCs required to implement a variety of radio waveforms.

5.2.1 BSP Creation

Unlike the other development platforms targeted in this dissertation, the Morpheus FPGAs do not have direct access to external memory. Instead, all memory accesses must originate from the ARM processor. The BSP for the Morpheus departed from this dissertation's configuration model as the partial bitstreams are not directly accessible to the FPGA. The Morpheus BSP targets the configuration controller code to the external ARM processor instead of an embedded MicroBlaze.

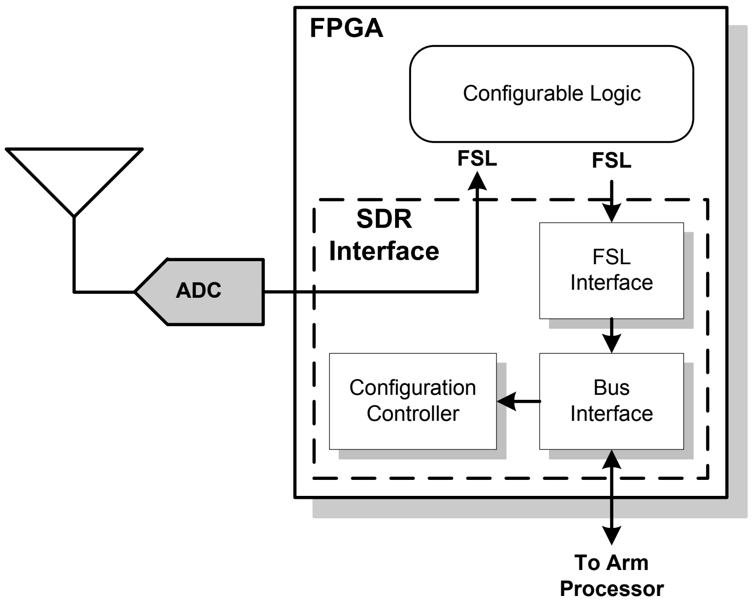


Figure 5.8: Morpheus BSP hardware components.

5.2.2 AM Radio Application

The AM interface module in the Morpheus BSP produces a stream of samples at 6 MHz. Before sampling by the ADC, the signal from the antenna is band-limited to 1.9 MHz by an analog low-pass filter to prevent aliasing during digitization. There are two primary approaches to AM demodulation: envelop detection and product detection. As product detection requires the generation of a phase and frequency matched sine wave to downconvert the RF signal to baseband, envelop detection was used instead.

Envelop detection leverages trigonometric identities to achieve down conversion of the signal from RF frequencies to baseband through a simple absolute value or squaring operation. As shown in Figure 5.9, the RF signal is band pass filtered to remove all but the desired station, downconverted to baseband via a squaring function, and decimated to reduce the sample rate from RF to audio frequencies.

Station tuning was achieved through filter swapping. Multiple band pass filters were created, one for each station, with partial reconfiguration placing the desired filter on the

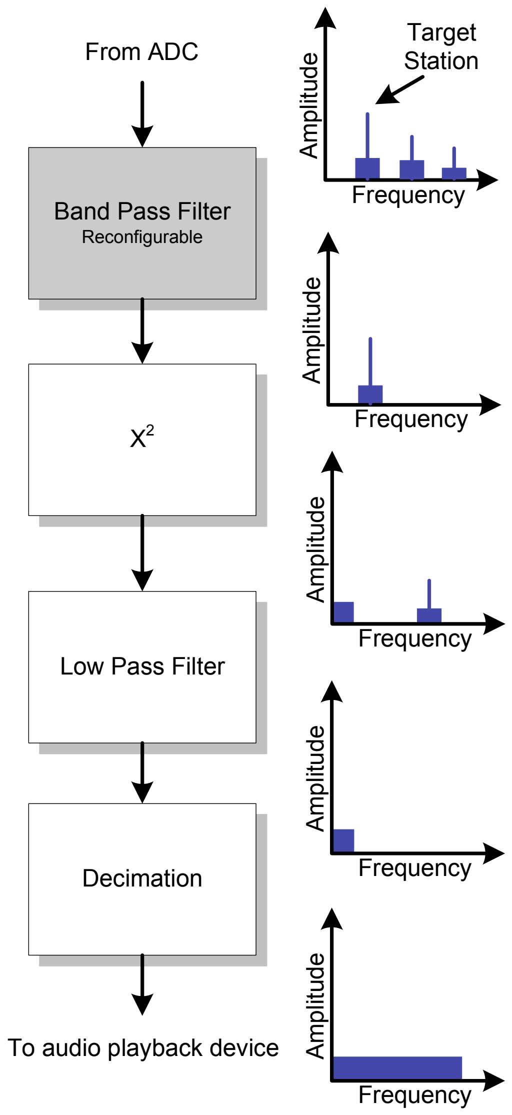


Figure 5.9: AM demodulation through envelop detection.

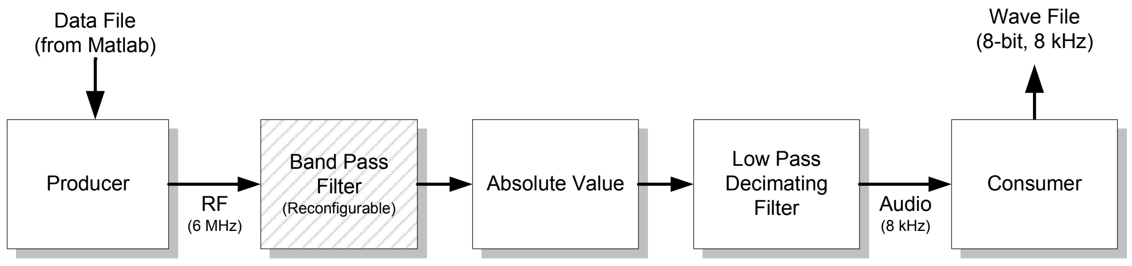


Figure 5.10: DR Impulse C simulation of AM demodulation.

radio. The demodulation scheme was first proved using Matlab. Test signals were created using data sampled from an antenna in the AM band. The stations in this test sample were frequency shifted to the same frequency as actual stations receivable by the Morpheus SDR. From this test set, filter coefficients for the bandpass and lowpass filters were created.

Simulations in Matlab are by default performed using floating point arithmetic. While floating point is preferable in many instances, it is not amenable to high-performance FPGA implementation. Because of this most signal processing, whether on FPGAs or DSPs, is performed using fixed-point math. Emulating real hardware, Impulse C uses integer math unless floating point variables are explicitly declared. The Impulse C simulation, see Figure 5.10, was a valuable resource to verify the Matlab filter coefficients in the presence of quantization affects caused by converting from floating point to fixed point.

Like the video processing application discussed above, the RCSF file generated for the SDR application had to be modified to remove the Impulse C testbench processes and replace them with the SDR Interface module from the Morpheus BSP, as can be seen by comparing Figures 5.11 and 5.12. Furthermore, due to the demanding performance constraints, the bandpass and lowpass filters were taken from the Xilinx CoreGen IP generation tool.

To effectively filter a single AM station, with its 10 kHz bandwidth, from a data stream sampled at 6 MHz required a 700 tap filter. If the filter ran at the same 6 MHz sampling frequency, 700 multipliers would be required, far more than is available on the device. However, the multipliers on the actual FPGA are capable of running much faster. By running the filter at 96 MHz and time-sharing a multiplier between several filter coefficients, the resource

```
<?xml version="1.0" ?>
<project implementation="Generic_VHDL" name="sdr_arch">
    <process folder="1430" module_name="fsl2bpf" name="bpf_process" type="static">
        <stream direction="input" name="waveform_raw" port_name="FSL0_S"/>
        <stream direction="output" name="waveform_filtered" port_name="FSL0_M"/>
    </process>
    <process module_name="sdr_if" name="SDR0" type="ip">
        <stream direction="input" name="baseband" port_name="FSL1_S"/>
        <stream direction="output" name="waveform_raw" port_name="FSL0_M"/>
    </process>
        ⋮
</project>
```

Figure 5.11: RCSF file for simulated SDR design.

```
<?xml version="1.0" ?>
<project implementation="Generic_VHDL" name="sdr_arch">
    <process folder="hw_bpf_process" module_name="fsl_bpf_process"
         name="bpf_process" type="static">
        <stream direction="input" name="waveform_raw" port_name="FSL0_S"/>
        <stream direction="output" name="waveform_filtered" port_name="FSL1_M"/>
    </process>
    <process folder="sw_consume_process" name="consume_process" type="sw">
        <stream direction="input" name="baseband" port_name="FSL0_S"/>
    </process>
    <process folder="sw_prod_process" name="prod_process" type="sw">
        <stream direction="output" name="waveform_raw" port_name="FSL0_M"/>
    </process>
        ⋮
</project>
```

Figure 5.12: RCSF file for implemented SDR design.

requirements are reduced to a point that the entire radio easily fits inside the FPGA. These multi-rate filters are readily available from the CoreGen tool, whereas to describe them in Impulse C requires additional forethought. An FSL-interface to the CoreGen-created filters was developed to facilitate use of existing IP. With these interfaces in hand, a user can describe the application in DR Impulse C while leveraging the full power of Xilinx-optimized filters, all without having to write a single line of HDL.

5.3 Benchmarking Applications

Two additional applications, an integer co-processor and a cryptographic accelerator, were created in the high-level development environment. Combined with the video processing and SDR applications, this set of demonstration designs validates the utility and productivity improvements afforded by this methodology. Furthermore, though tailored to streaming applications, the integer co-processor application shows that these techniques are applicable across other models of computation. The results of these benchmarking trials appear in the following chapter.

Chapter 6

Results

To quantify the productivity enhancements afforded by this development flow, a set of applications was implemented in this development environment and compared to implementations made manually using the Xilinx EAPR Flow. The differences in development time and application throughput are compared in the following sections.

This chapter concludes with a discussion of the results. The break-even point at which a PR application might be more advantageous than a static hardware application is presented for several applications. An analysis of the sources of area and performance penalties, as well as a discussion of mitigation techniques to improvement performance, follows.

6.1 Benchmark Applications

A coprocessor, cryptographic accelerator, and an SDR application have been benchmarked, providing three points of comparison. An additional comparison point was obtained by porting an existing application to a different FPGA architecture. These applications were selected as representative of the embedded, streaming applications where FPGAs are typically used.

6.1.1 Coprocessor Application

While the models of computation and communication selected by this dissertation favor streaming applications, the principles are applicable to other application domains. Coprocessor accelerators are frequently encountered in embedded applications. A small embedded processor may lack the performance required to meet the application's demands. By attaching a coprocessor tailored to a specific task, overall performance can be improved by off-loading demanding tasks to the coprocessor. The utility of an attached coprocessor increases with the number of functions it implements.

A reconfigurable coprocessor for an embedded MicroBlaze processor was created in this environment. This coprocessor, attached via an FSL interface, can be reconfigured to implement either a 32-bit integer divider or an integer square root function. Typical implementations of square root rely on the iterative CORDIC algorithm. To ease application development an existing software algorithm, using a look-up table for an initial estimate that is iteratively refined, was ported to Impulse C.

The descriptions for both functions were obtained from existing IP using the Xilinx Coregen tool and the OpenCores internet IP repository, in the case of the EAPR Flow, and using example code provided with the Impulse C tools, in the case of this project's development environment.

The development time for both environments, from initial design description to working hardware implementation, was recorded. In both cases significant portions of time were spent debugging. In the case of the Xilinx EAPR Flow troubleshooting was required of the control code that manages the ICAP configuration port, as well as the logic interfacing the CoreGen-created modules to the FSL interface. For this dissertation's high-level development environment the configuration code generation and FSL-interface logic generation were performed automatically, requiring no user intervention. However, issues with the hardware Impulse C generated for the division operation required an HDL simulation to successfully troubleshoot. A slight revision of the Impulse C description of this function resolved the

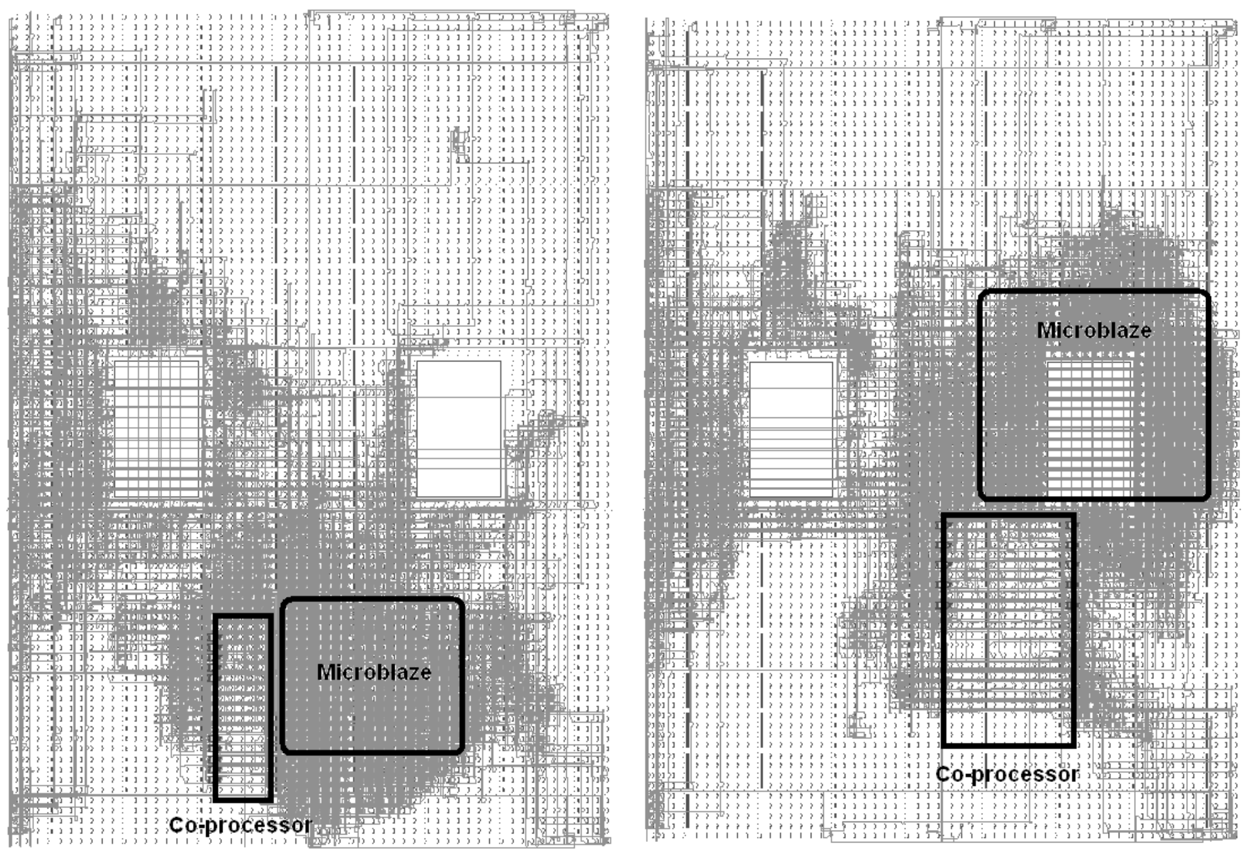


Figure 6.1: Co-processor implementation for designs created manually (left) and with development flow (right) on a Xilinx XC2VP30.

issues.

The layout of each implemented design is presented in Figure 6.1. The PR region of the Xilinx EAPR Flow was hand placed and is 36% smaller than the Impulse C-based approach, owing to inefficiencies in HLS and automated floorplanning.

Table 6.1 presents area and performance results at the module level. The Impulse C-generated divider compares well with the OpenCores divider, while the Coregen square root function is significantly smaller than the Impulse C-generated module. The Impulse C-generated square root function has a latency that is data-dependent. It should be noted that this high-level development environment can use existing IP and is not limited to Impulse C-created hardware. At the moment the implementation flow only supports IP with an FSL

Module	**Area** (Slices/BRAMs/BMults)	**Speed** (MHz)	**Throughput** (ops/sec)
Divider (Impulse C)	258 / 0 / 0	134	3.8 (10^6)
Divider (OpenCores)	159 / 0 / 0	123	3.4 (10^6)
Square Root (Impulse C)	760 / 1 / 9	56	0.7 (10^6) - 4.7 (10^6)
Square Root (CoreGen)	266 / 0 / 0	114	9.5 (10^6)

Table 6.1: Coprocessor module performance benchmarks on a Xilinx Virtex-II Pro.

interface.

As presented in Table 6.2, for the co-processor application the high-level development environment produced a 56% reduction in development time while incurring a 71% penalty in average throughput and a 8% overall area penalty when compared to a manual implementation in the Xilinx EAPR Flow. This throughput metric averages the best and worst case throughputs for the divider and square root modules. The manual EAPR implementation ran the co-processor at the system's 100 MHz clock rate. The high level development environment ran the co-processor at 80% of the synthesis-tool estimated clock rate for the slowest co-process module, which resulted in a 43 MHz clock.

The lines of code for the application is listed in Table 6.2. The high-level development approach required 33% fewer lines of code. With increasing code size comes increasing complexity, cost, and development time. It should be noted that the lines of code for the high-level development environment are written strictly in Impulse C, which is ANSI C compliant. The Xilinx EAPR code, on the other hand, is a mixture of HDL and C, for the controller. Mixed language designs further increase complexity and mandate additional skill requirements. Furthermore, the Impulse C lines of code metric includes a high-level testbench, in addition to the application description. The EAPR metric excludes any HDL testbench.

The performance penalty could be reduced by leveraging existing IP instead of using Impulse C-generated HDL. Additional gains are possible by dynamically modifying the clock rate of the co-processor instead of running the all co-processors at the speed of the slowest.

Environment	**Area** (slices)	**Ave. Throughput** (ops/sec)	**Lines of Code**	**Dev. Time** (hrs)
High Level (Impulse C)	3,118	1.6 (10^6)	293	6
Xilinx EAPR	2,883	5.6 (10^6)	439	13.5

Table 6.2: Coprocessor productivity and performance benchmarks.

The small area penalty is due to the superiority of hand-placed designs.

6.1.2 Cryptographic Accelerator

Cryptographic hash functions transform a string of bits comprising a message into a fixed-length number, called the message digest. These functions have the desirable property that any small change in the message will result in a completely different message digest. Hash functions are typically used to verify that data was transmitted without errors or modifications.

A cryptographic accelerator was described in DR Impulse C that implements either an MD5 hash or a SHA-1 hash. The DR Impulse C description was developed from scratch. While C code for both of these functions is widely available, most implementations utilize data types and functions are that are not directly supported in Impulse C. Furthermore, the development of an involved application from start to finish aided in benchmarking the development environment.

In this application a reconfigurable region on the FPGA could be configured for either the MD5 or the SHA-1 standard. The hash functions were created from scratch using both Impulse C and Verilog. Area and performance numbers for each function are shown in Table 6.3. The Verilog-described SHA-1 consumed 12% more slices than the Impulse C design owing to the use of five independent memories to permit simultaneous access to the message data. This approach increases throughput at the expense of area. Had area been of primary concern a Verilog design would have been smaller than the Impulse C-created hardware. The Impulse C MD5 and SHA-1 cores underperformed the Verilog cores by 39% and 63%,

Module	**Area** (Slices/BRAMs)	**Speed** (MHz)	**Throughput** (blocks/sec)
MD5 (Impulse C)	1305 / 2	66	0.43 (10^6)
MD5 (Verilog)	613 / 0	61	0.71 (10^6)
SHA-1 (Impulse C)	1080 / 1	73	0.17 (10^6)
SHA-1 (Verilog)	1214 / 0	76	0.46 (10^6)

Table 6.3: Cryptographic module performance benchmarks in a Xilinx Virtex-II Pro.

Environment	**Area** (slices)	**Throughput** (blocks/sec)
High Level (Impulse C)	4,574	0.25 (10^6)
Xilinx EAPR	4,107	0.53 (10^6)

Table 6.4: Cryptographic application performance benchmarks.

respectively.

Table 6.4 presents the performance results with the cryptographic modules integrating into the reconfiguration application including the configuration controller. The high-level development environment imparts a 11% overall area increase and a 53% performance penalty, compared to the conventional Verilog design.

The productivity advantage of the high-level development environment was hampered in this application by a bug in the Impulse C-generated hardware, as seen in Table 6.5. The HLS-generated hardware's barrel shifter, required by the MD5 hash, used incorrect VHDL datatypes for synthesis. While a workaround was identified from a support forum, the additional time spent troubleshooting this and another issue with the MD5 hash resulted in a 28% greater frontend design time for the high-level development environment than for a Verilog-created design. If the MD5 design time was removed from consideration, the frontend design times for the high-level and the conventional approaches are 1.0 and 2.2 hours, respectively. This 55% frontend design time improvement is more in line with the co-processor productivity results. If the MD5 design and debug time is considered the total development improvement of the high-level approach is 10%, while if the MD5 design time is excluded from both designs the high-level productivity improvement increases to 49%,

Environment	**Frontend**		**Backend**	**Total**		**Lines of Code**
	(hrs)	(w/o MD5)	(hrs)	(hrs)	(w/o MD5)	
High Level	8.1	1.0	3.3	11.3	4.3	472
Xilinx EAPR	6.3	2.2	6.3	12.5	8.5	593

Table 6.5: Cryptographic application productivity benchmarks.

approximating the results for the co-processor application. In terms of lines of code, the high-level development approach required 20% fewer lines of code to describe.

6.1.3 Application Porting

The initial design effort required to deploy an application is an important part of an overall productivity metric. An additional component of productivity is the effort required to port an existing application to a new platform. To evaluate the utility of this dissertation to application porting, the cryptographic accelerator, originally created on a Xilinx Virtex-II Pro FPGA was ported to a Virtex-4 FPGA.

The differences between these two devices for RTR applications is quite large. The Virtex-4 incorporates an updated configuration architecture with more granular configuration frames. The ICAP port on a Virtex-4 is capable of operating in a 32-bit mode, as opposed to the 8-bit mode of the Virtex-II Pro, speeding reconfiguration but complicating partial bitstream creation. The locations and structures of dedicated memories and multipliers is different between the two architectures. These differences necessitate the creation of a new configuration controller and floorplan, as well as a reimplementation of all modules.

The results, summarized in Table 6.6, indicate that the high-level development environment imparts a 14% area penalty and a 74% performance penalty compared to a manual implementation with the EAPR. The performance difference between the two flows for the cryptographic accelerator is much larger in the Virtex-4 architecture than the Virtex-II Pro owing to an increased sensitivity of timing to placement. The critical path that determines the clock frequency at which a reconfigurable module runs was found to be in the FSL con-

Environment	**Area** (slices)	**Throughput** (blocks/sec)	**Time to Port** (hrs)
High Level (Impulse C)	5,083	0.14 (10^6)	0.7
Xilinx EAPR	4,392	0.53 (10^6)	3.0

Table 6.6: Cryptographic application performance benchmarks ported to Xilinx Virtex-4.

trol and status signals for the Virtex-4. These signals originate in the reconfigurable module and must pass through the bus macros to reach the FSL FIFO control logic, after which the signal is used to produce a status flag that is sent back through the bus macros to arrive at the reconfigurable module before the next rising clock. Due to the differences between the Virtex-II Pro and Virtex-4 FPGAs, the timing on these paths is more critical in the Virtex-4, as they did not appear prominently in the timing reports of Virtex-II Pro designs. While far from optimal, the chosen solution was to modify the timing generation tool to reduce the clock frequency for reconfigurable modules in Virtex-4 FPGAs. The reconfigurable modules' clock frequency for the automated flow was 31 MHz, compared to 60 MHz for the manually placed design. Alternative solutions, such as improving the quality of the floorplanner tool's placements or the addition of a latency-independent interface, are left to future work.

The productivity improvement of the high level development environment was found to be 77%, in line with the implementation time improvements seen in the other applications.

6.1.4 Software Defined Radio Application

The AM radio application targeting the Morpheus SDR platform was implemented in both the high-level development environment and the Xilinx EAPR Flow. The results provided in Table 6.7 only record the implementation time and exclude the frontend design effort. As described in Section 5.2.2, Impulse C was leveraged in the radio's design to understand the quantization effects of fixed-point arithmetic. Because this high-level development tool was utilized in the initial design of both the EAPR and HLS radios, it was not possible to separate the time spent in the frontend design stage for each flow.

Limitations in Impulse C's HLS required the use of pre-existing filters. While capable of producing high-performance filters, the large number of taps in the AM radio's band pass filter appear to be beyond the capabilities of the tool. A multirate filter could be realized by Impulse C to meet the required specifications at the expense of increased design complexity. As GUI-customized, architecture-optimized filters are available from all the major FPGA vendors, it is reasonable to assume that a design engineer would leverage these existing cores rather than accept a performance penalty for HLS. Furthermore, the AM radio application demonstrates the applicability of this high-level development approach to typical IP reuse scenarios. By leveraging existing IP the development can be further reduced.

The FPGAs in the Morpheus development platform do not have direct access to external storage. Partial bitstreams are pushed to the FPGA from an ARM processor. As the high-level development environment expects an embedded configuration controller, hand editing of the automatically created top level Verilog wrapper was required to properly connect the reconfigurable module's reset and bus macro enable signals.

The floorplan created by the automated Floorplanner tool required manual tweaking to correct a user mistake in the creation of an XML file specifying the module's resource requirements. This manual editing resulted in both flows using the same area.

While the lines of code required to describe the application in each development environment were similar, it should be noted that the Impulse C description includes the high-level testbench that reads in real RF data from a file and produces an audio WAV file for verification.

The implementation time in Table 6.7 refers to the time it took to transition from the initial static design to a partially reconfigurable design. The automation provided by the high-level approach reduced implementation time by 70%, largely due to the elimination of the errors that are inevitable in the design process.

Environment	**Area** (slices)	**Throughput** (Msamples/sec)	**Implement. Time** (hrs)	**Lines of Code**
High Level	2,684†	6.0	0.8	515
Xilinx EAPR	2,685	6.0	2.7	520

†Hand-crafted area constraints.

Table 6.7: AM radio application performance benchmarks on the Morpheus SDR platform.

6.2 Conclusions

Figure 6.2 graphically displays the results from the preceding section. The results for area, performance, and development time for this dissertation's high-level development environment are normalized to those of the EAPR flow for each benchmark application. As can be seen in the figure, the area penalty of a high-level development approach is minimal. Performance, in terms of throughput, does suffer notably, but the performance penalty closely mirrors the productivity gains. Note that the SDR application results differ somewhat from this trend, as the performance of both radio implementations met the minimum required throughput.

It should be noted that these results do not take into account the additional skill set required to even attempt an RTR application using the Xilinx EAPR Flow. The vendor-provided documentation assumes a detailed knowledge of the Xilinx implementation tools and of the Virtex FPGA architecture. Furthermore, little vendor support is available to resolve RTR design issues. The numbers present in Figure 6.2 were obtained by an experienced designer intimately familiar with Xilinx FPGA tools and architectures.

In comparison, this dissertation's high-level development environment requires only a working knowledge of Impulse C, though a user with hardware design experience would produce improved results. As Impulse C leverages standard ANSI C, the pool of potential users is orders of magnitude larger than that of the Xilinx EAPR flow.

The area and performance penalties introduced by this dissertation's development envi-

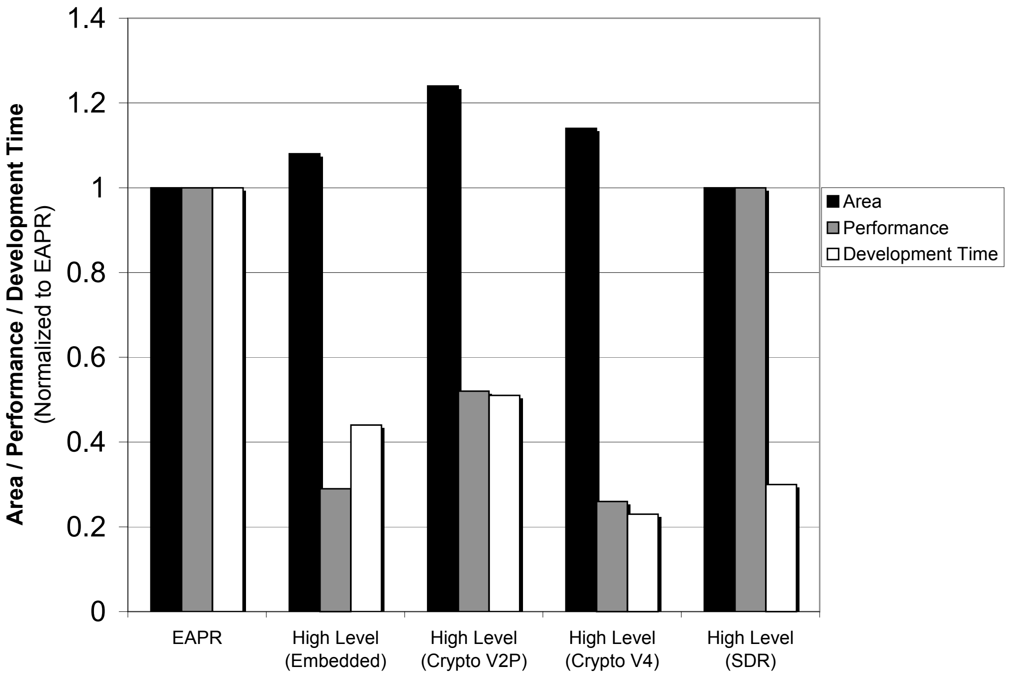


Figure 6.2: Comparison of EAPR flow to high level development environment for normalized area, performance, and development time for applications on the Virtex-II Pro (V2P) and Virtex-4 (V4).

ronment, compared to the Xilinx EAPR flow, come from three sources:

1. HLS inefficiencies. The hardware generated by any current HLS tool will be larger and slower than hardware that an experienced designer could produce. These penalties have been documented by others [52] [84] and the cryptographic accelerator results are in line with the published research.

2. Conservative floorplans. Currently, the automated implementation flow is not closed-loop. A partially reconfigurable region that is sized too small to hold a module will result in an error that requires user intervention. The floorplanner tool therefore sizes the reconfigurable regions conservatively, slightly increasing the area penalty.

3. Conservative clocking estimates. As in the case of the automated floorplanner, the clock generation algorithm conservatively selects a lower clock speed than might be acceptable to avoid a design failing timing analysis.

Of these sources, the HLS tool is the largest contributor to the area increases and throughput reductions. While directly addressing HLS tool inefficiencies is beyond the scope of this dissertation, some avenues do exist to improve performance using the current tools. Many HLS tools, Impulse C included, feature advanced directives to provide guidance as to the structure of the generated hardware. As productivity was weighted higher than performance for application benchmarking, only basic directives, such as which loops to pipeline, were provided to the HLS tool. By experimenting with multiple optimizations it is likely the penalties of the HLS tool could be reduced at the expense of development time.

The final two sources of area and performance penalties, the conservative floorplanning and clocking estimates, can be address by making the automated implementation flow closed-loop. Any design that is unroutable because of poorly sized area constraints could be re-floorplanned using a larger estimate of the resource requirements. Similarly, a design that misses timing could have its clocks adjusted to the fastest obtainable speed.

The benchmark applications permit an analysis of the utility of partial reconfiguration to be made. Figure 6.3 shows the area savings, in percent of slices, that partial reconfiguration provides for the number of modules that can be swapped in and out of hardware. This figure compares the area of a completely static design, where all of the modules is implemented in hardware at all times, to a partially reconfigurable design, where only one a single module is resident in the hardware at a time. In terms of area savings only, the break-even point where a completely static design uses the same amount of resources as a partially reconfigurable design is indicated on the figure as the location of a line's X-axis crossing. For the embedded coprocessor application this break-even point occurs between one and two reconfigurable modules. Any additional modules added to the system increase the area savings of partial reconfiguration. The area of a module in the figure was taken to be the average of the areas of the implemented modules for that application.

The equation of the area savings lines in the figure is given below, where N is the number of modules and $A_{savings}$ is in percent. The break-even point can be found by setting $A_{savings}$

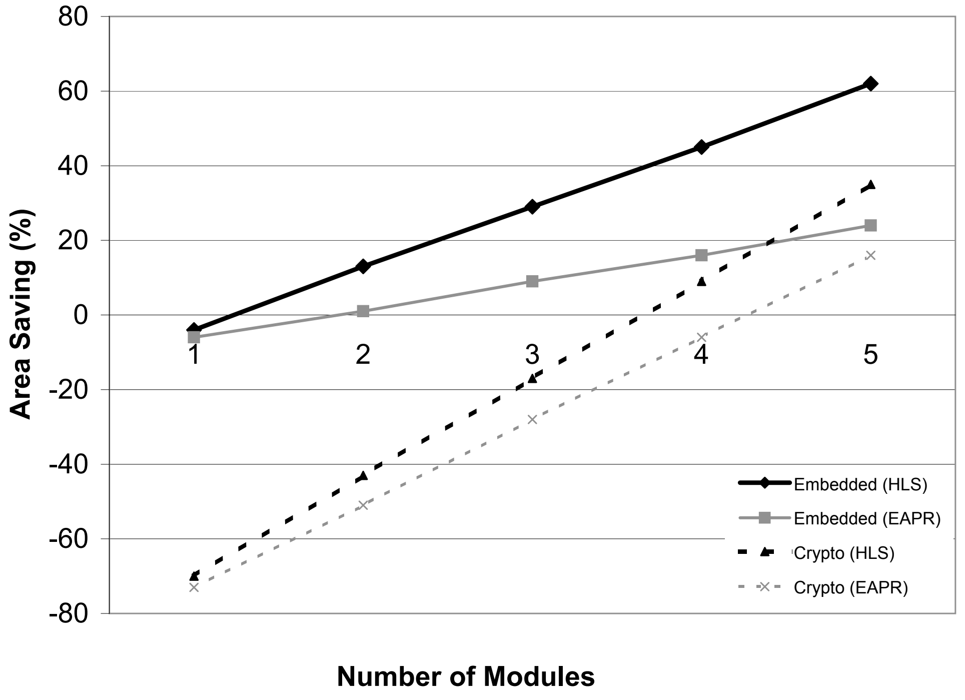


Figure 6.3: Area savings from partial reconfiguration.

to zero and solving for N.

$$A_{savings} = ((A_{static} + N * A_{module})/(A_{static} + A_{overhead} + A_{pr\_region}) - 1) * 100$$

Note that this figure presents a larger area savings for HLS designs compared to the EAPR flow, as evident by the higher slope of the HLS lines. This is because the area savings for the HLS applications are calculated based on the larger HLS reconfigurable modules.

As is evident from the figure, there is a large discrepancy in the area savings of the two applications. The embedded coprocessor application requires an embedded processor be integrated into the design. This embedded processor can also handle the configuration management of the device. For the cryptographic accelerator, however, no embedded processor is required to perform the hash function. The addition of a MicroBlaze processor to manage the reconfiguration adds to the area overhead of partial reconfiguration. Thus, partial reconfiguration is more advantageous for applications that already require an embedded processor.

Design	Cost [85]	Static Power [86]
PR HLS (XC4VLX60-10)	$635	487 mW
Static HLS (XC4VLX25-10)	$220	227 mW

Table 6.8: Cost and power benefits of partial reconfiguration

If the architecture incorporates a hard core processor, such as the PPC405 found in Xilinx Virtex-II Pro and Virtex-4 FX devices, this processor can supply configuration management with a reduced area penalty.

To better quantify the benefits of partial reconfiguration it is instructive to consider actual device cost and power consumption. Because larger die are more prone to defects, FPGA device costs increase non-linearly with device size. For an application with the same area savings as the embedded application in Figure 6.3, if it is assumed that the static design would fit inside a Xilinx XC4VLX60 FPGA, than the dynamically reconfigurable design could be placed within a XC4VLX25. Table 6.8 presents the cost and area benefits for such an application. By using partial reconfiguration the FPGA device cost is reduced by 65% and the static power consumption is reduced by 53%.

The experience of benchmarking multiple applications revealed reoccurring implementation issues common to both the high-level development environment and the Xilinx EAPR flow. Addressing the following issues would improve performance and productivity for all RTR applications:

- Standardized configuration interfaces. In spite of the fact that the ports on the Xilinx ICAP remain similar across the architectures, the timing of the signals varies across the devices and the interface cores provided by Xilinx change between versions of their tools. As was experienced in the creation of the BSPs for the high-level development environment and the manual implementation of a configuration controller using the Xilinx EAPR flow, the differences in the ICAP and its interface cores can greatly complicate system integration. Furthermore, debugging of any configuration issues frequently requires the modification of the design to insert logic analyzer probes directly to the ICAP inputs and outputs. No other component of a Xilinx FPGA requires such effort to utilize. The creation of a static, well-defined interface to the ICAP that is portable across architectures and tool versions would facilitate the implementation of RTR designs.

- Dynamic clock frequency modifications. Currently, in both the high-level development environment and the Xilinx EAPR flow, the clock frequency of a reconfigurable module is locked to a single fixed value at compile time. This reduces the clock frequency of a reconfigurable module to the frequency of the slowest module that might be placed on the device. However, dynamic modification of a module's clock frequency can be achieved through partial reconfiguration of the FPGA's clock managers or through the use of a dynamic reconfiguration port available on the clocking resources of certain FPGAs. In the case of the embedded coprocessor application, this dynamic clock modification could have increased the throughput of the integer divider by a factor of two.

- Latency-independent communication schemes. As previously discussed, in many designs implemented on the Virtex-4 the critical timing path is in status and control logic in the FSL communication link. To improve perform of the designs and significantly reduce the sensitivity of performance to a design's floorplan, a latency-independent communication scheme should be adopted. FSL links, as well as many busses, generally require that a module respond quickly to a control or status event. A communication scheme that was more tolerant of delays would remove the communication interconnects from the critical timing path, thus increasing clock frequency and throughput.

These benchmark results validate this high-level development methodology, while suggesting avenues for improvement. In real-world scenarios the benefits of this approach would likely be greater, as these tools empower designers with limited hardware experience to tackle challenging dynamic hardware applications.

Chapter 7

Conclusion

This dissertation has defined a comprehensive approach to dynamic hardware application development. Models of computation, communication, and reconfiguration are specified appropriate to streaming applications. Through the creation of configuration management functions, the low-level details of partial reconfiguration are hidden from a designer. The specified design flow is independent of a specific language and design environment, permitting the development environment to fit the application. As HDLs cannot describe the dynamic modification of communication or computation structures, a special specification format augments the HDL, describing the configuration controller, dynamic hardware, and other design aspects not captured elsewhere. This RTR computing specification links the frontend design flow to the backend implementation flow. By separating the design from the implementation, applications can be easily ported across different architectures.

This dissertation has created an end-to-end design and implementation environment, permitting, for the first time, the high-level development of autonomous, partially reconfigurable applications. The open nature of the design and implementation flows enables other projects to extend this work to encompass additional high-level design capture environments, FPGA architectures, and reconfiguration strategies.

Through the creation of a set of benchmarking application, the performance and area penalties of this high-level design environment were compared to those obtained by a traditional, manual implementation. The results indicate that for a modest area overhead, the high-level design approach can reduce development time by 50% or more. While a performance penalty on the order of 50% to 70% percent was observed, in many applications the reduced design time would make this approach attractive, particularly in light of the significantly reduced skill set that this high-level development approach affords.

Unlike previous attempts to raise the level of design abstraction for partially reconfigurable applications, this dissertation has produced a complete design and implementation flow, never requiring the designer to depart from HLLs. The extensible nature of this development environment facilitates the addition of new architectures and design entry techniques. The automated inclusion of a configuration controller opens the domain of embedded computing to the benefits of dynamically reconfigurable hardware.

The results validate the benefits of partial reconfiguration. By reducing the area requirements, cost and power consumption are reduced. While the overheads associated with partial reconfiguration, in terms of a configuration controller, and HLS, in terms of area inefficiencies, substantial improvements in cost and power are seen for designs that can share hardware resource between four or more reconfigurable modules.

While demonstrated with FPGAs, the relevance of this work extends to other domains that leverage the CSP model and permit dynamic modification of computational operations. Multiple-processor systems operated in a streaming application paradigm can leverage these models and techniques, as can certain coarser-grained configurable architectures.

Multiple avenues exist for expanding the breadth and depth of this work, including:

- Removal of Xilinx EAPR Flow constraints. This dissertation builds off of the Xilinx EAPR Flow and shares some of its limitations. A partial bitstream can only be placed in a single reconfigurable region communicating over fixed interconnections. A related project recently demonstrated the ability to relocate partial bitstreams while rerouting the signals coming to and from the modules [77]. Integration of these tools into this work would increase its utility for certain application domains, such as SDR, by enabling dynamic topologies.

- Integration of debugging and verification tools. HLS tools generally do not provide a direct mechanism for simulating the design at the HDL level. In many instances no HDL simulation is required. However, for debugging an HLS-generated design, HDL simulation is invaluable. By automating the creation of an HDL testbench from the user's HLL testbench, the process of verifying the HDL against the user's HLL description can be automated. This automatic verification can be extended to the hardware as well. The streams connecting CSP modules are excellent observation and comparison points, as the data that passes through the streams should match at each level of abstraction – executing hardware, HDL simulation, and HLL simulation.

- Cross-platform support. The CSP computation model can encompass multi-core processing quite easily. For certain applications, such as floating-point arithmetic, a General Purpose Processor (GPP) is preferable to an FPGA. By expanding the environment to include GPPs, the applicability of this dissertation can encompass other computational domains.

Bibliography

[1] I. Verbauwhede and P. Schaumont, "The happy marriage of architecture and application in next-generation reconfigurable systems," in *CF '04: Proceedings of the 1st conference on Computing frontiers*, (New York, NY, USA), pp. 363–376, ACM Press, 2004.

[2] E. Lemoine and D. Merceron, "Run time reconfiguration of fpga for scanning genomic databases," in *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, (Washington, DC, USA), p. 90, IEEE Computer Society, 1995.

[3] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet processing on the field programmable port extender (FPX)," in *FPGA*, pp. 87–93, 2001.

[4] P. James-Roxby and B. Blodget, "Adapting constant multipliers in a neural network implementation," in *FCCM'00: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Washington, DC, USA), p. 335, IEEE Computer Society, 2000.

[5] D. I. Lehn, R. D. Hudson, and P. M. Athanas, "Framework for architecture-independent run-time reconfigurable applications," *Reconfigurable Technology: FPGAs for Computing and Applications II*, vol. 4212, no. 1, pp. 162–172, 2000.

[6] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures," in *IPPS/SPDP Workshops*, pp. 31–36, 1998.

[7] K. Bondalapati, P. Diniz, P. Duncan, J. Granack, M. Hall, R. Jain, and H. Ziegler, “Defacto: a design environment for adaptive computing technology,” in *Parallel and Distributed Processing. 11th IPPS/SPDP’99 Workshops*, (Berlin, Germany), 1999.

[8] A. Rudra, “The rising importance of FPGA technology in software defined radio,” *COTS Journal*, January 2005. January 2005.

[9] A. Malagamba, “SDR prêt-á-porter,” *FPGA and Structured ASIC Journal.* February 28th, 2006.

[10] J. Villasenor and W. Mangione-Smith, “Configurable computing,” *Scientific American*, pp. 66–71, June 1997.

[11] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, “A quick safari through the reconfiguration jungle,” in *Design Automation Conference*, pp. 172–177, 2001.

[12] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, “Reconfigurable computing: architectures and design methods,” *Computers and Digital Techniques, IEE Proceedings-*, vol. 152, no. 2, pp. 193–207, 2005.

[13] R. Tessier and W. Burleson, “Reconfigurable computing for digital signal processing: A survey,” *Journal of VLSI Signal Processing*, vol. 28, pp. 7–27, June 2001.

[14] K. Jarvinen, M. Tommiska, and J. Skytta, “Comparative survey of high-performance cryptographic algorithm implementations on fpgas,” in *Information Security, IEE Proceedings*, vol. 152, pp. 3–12, 2005.

[15] T. Ramdas and G. Egan, “A survey of FPGAs for acceleration of high performance computing and their application to computational molecular biology,” in *Proceedings of TENCON*, pp. 1–6, Nov 2005.

[16] P. Athanas and A. Abbott, “Real-time image processing on a custom computing platform,” *Computer*, vol. 28, no. 2, pp. 16–25, 1995.

[17] V. Pratt, "Anatomy of the Pentium Bug," in *TAPSOFT'95: Theory and Practice of Software Development* (P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, eds.), no. 915, pp. 97–107, Springer Verlag, 1995.

[18] E. Lechner and S. Guccione, "The Java environment for reconfigurable computing," in *7th International Workshop on Field Programmable Logic and Applications, FPL*, pp. 284–293, 1997.

[19] M. J. Wirthlin and B. L. Hutchings, "Improving functional density through run-time constant propagation," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (Monterey, CA), pp. 86–92, 1997.

[20] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *FCCM'97: IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 77–86, 1997.

[21] W. Luk, N. Shirazi, and P. Cheung, "Modelling and optimising run-time reconfigurable systems," in *FCCM'06: IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 167–176, IEEE Computer Society Press, 1996.

[22] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java based interface for reconfigurable computing," in *Proceedings of Military and Aerospace Applications of Programmable Logic Devices Conference (MAPLD)*, (Laurel, Maryland), 1999.

[23] R. Lysecky, F. Vahid, and S. Tan, "Dynamic fpga routing for just-in-time fpga compilation," in *Design Automation Conference*, 2004.

[24] K. Chia, H. Kim, S. Lansing, W. Mangione-Smith, and J. Villasenor, "High-performance automatic target recognition through data-specific VLSI," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 364–371, 1998.

[25] D. Ross, O. Vellacott, and M. Turner, "An FPGA-based hardware accelerator for image processing," in *Selected papers from the Oxford 1993 international workshop on field*

programmable logic and applications on More FPGAs, (Oxford, UK, UK), pp. 299–306, Abingdon EE&CS Books, 1994.

[26] A. Rashid, J. Leonard, and W. H. Mangione-Smith, "Dynamic circuit generation for solving specific problem instances of boolean satisfiability," in *FCCM'98: IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 196–204, 1998.

[27] M. J. Wirthlin and B. L. Hutchings, "Sequencing run-time reconfigured hardware with software," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (Monterey, CA), pp. 122–128, 1996.

[28] C. Patterson, "High performance DES encryption in Virtex FPGAs using JBits," in *FCCM'00: IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 113–121, 2000.

[29] C. Cox and W. Blanz, "Ganglion—a fast field-programmable gate array implementation of a connectionist classifier,," *IEEE Journal of Solid-state Circuits*, vol. 27, no. 3, pp. 288–299, 1992.

[30] A. Poetter, "JHDLBits: An open-source model for fpga design automation," Master's thesis, Virginia Polytechnic and State University, 2004.

[31] B. Blodget, S. McMillan, and P. Lysaght, "A lightweight approach for embedded reconfiguration of FPGAs," in *Proceedings of the Design,Automation and Test in Europe Conference and Exhibition*, 2003.

[32] J. G. Eldredge and B. L. Hutchings, "RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs," vol. 4, pp. 2097–2102 vol.4, 1994.

[33] J. D. Hadley and B. L. Hutchings, "Design methodologies for partially reconfigured systems," pp. 78–84, 1995.

[34] J. Villasenor, C. Jones, and B. Schoner, "Video communications using rapidly reconfigurable hardware," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 6, pp. 565–567, 1995.

[35] "FALCON II global family of products," product brief, RF Communications Division, Harris Corp., 2006.

[36] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[37] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11–18, 1993. 0018-9162.

[38] "S5530 software-configurable processor," product brief, Stretch, Inc., 2006.

[39] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors (ITRS)."

[40] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, 2003.

[41] R. Goering, "High-level synthesis rollouts enable ESL," *EETimes.* May 31, 2004.

[42] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 11, no. 4, pp. 44–54, 1994.

[43] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *FCCM'00: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 49–56, 2000.

[44] A. Brito, M. Khnle, M. Hbner, J. Becker, and E. Melcher, "Modelling and simulation of dynamic and partially reconfigurable systems using SystemC," in *processing of IEEE Computer Society Annual Symposium on VLSI*, pp. 35–40, 2007.

[45] A. Pelkonen, K. Masselos, and M. Cupak, "System-level modeling of dynamically reconfigurable hardware with systemc," in *processing of International Parallel and Distributed Processing Symposium*, 2003.

[46] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, "Automatic mapping of khoros-based applications to adaptive computing systems," in *Proceedings of Military and Aerospace Applications of Programmable Devices Conference (MAPLD)*, (Washington, D.C.), 1999.

[47] Celoxica, Inc., "Handel-C for hardware design," white paper, 2006.

[48] D. Pellerin and S. Thibault, *Practical FPGA Programming in C.* Upper Saddle River, N.J.: Prentice Hall, 2005.

[49] Nallatech, "Dimetalk v3.0 application development environment," product brief, 2006.

[50] D. Poznanovic, "Application development on the SRC Computers, Inc. systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[51] Xilinx, Inc., "Xilinx System Generator for DSP version 8.2," user's guide, 2006.

[52] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, and A. Troxel, I. George, "Survey of c-based application mapping tools for reconfigurable computing," in *Proceedings of Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, (Washington, D.C.), 2004.

[53] R. D. Hudson, D. Lehn, J. Hess, J. Atwell, D. Moye, K. Shiring, and P. Athanas, "Spatio-temporal partitioning of computational structures onto configurable computing machines," in *Configurable Computing: Technology and Applications, Proc. SPIE 3526* (J. Schewel, ed.), (Bellingham, WA), pp. 62–71, SPIE – The International Society for Optical Engineering, 1998.

[54] M. Eisenring and M. Platzner, “A framework for run-time reconfigurable systems,” *J. Supercomput.*, vol. 21, no. 2, pp. 145–159, 2002.

[55] E. Carvalho, N. Calazans, E. Briao, and F. Moraes, “Padreh - a framework for the design and implementation of dynamically and partially reconfigurable systems,” pp. 10–15, 2004.

[56] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, “Stream computations organized for reconfigurable execution (SCORE),” in *Field-Programmable Logic and Applications*, (Berlin, Germany), 2000.

[57] F. Ferrandi, M. Santambrogio, and D. Sciuto, “A design methodology for dynamic reconfiguration: the caronte architecture,” *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 4 pp.–, 4-8 April 2005.

[58] A. Antola, M. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, “A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper,” *3rd Southern Conference on Programmable Logic, (SPL)*, pp. 221–224, 28-26 Feb. 2007.

[59] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad, “Model-integrated tools for the design of dynamically reconfigurable systems,” technical report, Institute for Software Integrated Systems, Vanderbilt University, 2000.

[60] W. Luk, N. Shirazi, and P. Y. K. Cheung, “Compilation tools for run-time reconfigurable designs,” in *FCCM ’97: IEEE Symposium on FPGA-Based Custom Computing Machines*, (Washington, DC, USA), p. 56, IEEE Computer Society, 1997.

[61] T. K. Lee, A. Derbyshire, W. Luk, and P. Y. K. Cheung, “High-level language extensions for run-time reconfigurable systems,” in *Field-Programmable Technology (FPT). Proceedings. IEEE International Conference on*, pp. 144–151, 2003.

[62] A. L. Slade, B. E. Nelson, and B. L. Hutchings, "Reconfigurable computing application frameworks," in *FCCM'03: IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 251–260, 2003.

[63] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," *EURASIP Journal on Embedded Systems*, 2007.

[64] E. A. Lee and T. M. Parks, "Dataflow process networks," in *Proceedings of the IEEE*, pp. 773–799, May 1995.

[65] C. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[66] P. Welch and D. Wood, "The Kent Retargetable occam Compiler," in *Parallel Processing Developments, Proceedings of WoTUG 19*, vol. 47, pp. 143–166, Mar. 1996.

[67] P. Ljung, "How to create fixed- and floating-point IIR filters for FPGAs," *Programmable Logic Design Line.* May 31, 2006.

[68] Formal Systems, "Failures-divergence refinement," user manual, 2005.

[69] E. Lee, "Overview of the ptolemy project," summary paper, 2003. Available at http://ptolemy.eecs.berkeley.edu/publications/papers/03/overview/overview03.pdf.

[70] A. Saifhashemi and P. Beerel, "High level modeling of channel-based asynchronous circuits using Verilog," in *Communicating Process Architectures Conference*, (Eindhoven, Netherlands), 2005.

[71] Xilinx, Inc., "Microblaze processor reference guide," reference manual, 2006.

[72] J. A. Williams, N. W. Bergmann, and X. Xie, "FIFO communication models in operating systems for reconfigurable computing," in *FCCM'05: IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 277–278, 2005.

[73] Xilinx, Inc., “AccelDSP synthesis tool,” product information, 2006. Available at http://www.xilinx.com/ise/dsp_design_prod/acceldsp/index.htm.

[74] I. Xilinx, “XAPP290: Two flows for partial reconfiguration: Module based or difference based,” 2004.

[75] S. Koh and O. Diessel, “Comma: A communications methodology for dynamic module-based reconfiguration of fpgas,” in *proceedings of the Dynamically Reconfigurable Systems Workshop in the International Conference of Architectures of Computing Systems*, 2006.

[76] C. Bobda and B. Ali Ahmadinia, “Dynamic interconnection of reconfigurable modules on reconfigurable devices,” *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 443–451, 2005.

[77] P. Athanas, J. Bowen, T. Dunham, C. Patterson, M. Rice, J. Shelburne, J. Suris, M. Bucciero, and J. Graf, “Wires on demand: Run-time communication synthesis for reconfigurable computing,” in *International Conference on Field Programmable Logic and Applications, FPL*, 2007.

[78] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” in *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.

[79] J. Newcomb, “A scalable approach to multi-core prototyping,” master, Virginia Polytechnic Institute and State University, 2008.

[80] R. Stallman, R. McGrath, and P. Smith, *The GNU Make Manual.* Boston, MA: Free Software Foundation, 2006.

[81] L. Cheng and M. Wong, “Floorplan design for multimillion gate FPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System*, vol. 25, no. 12, pp. 2795–2805, 2006.

[82] Y. Feng and D. Mehta, "Heterogeneous floorplanning for FPGAs," in *Proceedings of the 19th International Conference on VLSI Design (VLSID06)*, (Washington, DC, USA), IEEE Computer Society, 2006.

[83] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *International Workshop on Field Programmable Logic and Applications, FPL*, 2006.

[84] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, 28-26 Feb. 2007.

[85] Avnet, Inc., "Avnet partbuilder," price quote, May 15, 2008.

[86] Xilinx, Inc., "Xilinx power estimator user guide," reference manual, 2008.