

Structured Approach to Dynamic Computing Application Development

Stephen Douglas Craven

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

Doctor of Philosophy
in
Electrical Engineering

Dr. Peter M. Athanas, Chair

Dr. Cameron D. Patterson

Dr. Patrick R. Schaumont

Dr. Gary S. Brown

Dr. Shawn A. Bohner

October 2006

Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

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

Copyright 2006 ©, Stephen Douglas Craven

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. The proposed research 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 may be hidden, permitting a designer to focus instead on a design's behavior. The proposed approach frees reconfigurable applications from dependence on an external configuration controller, generating a configuration manager from a high-level description. The proposed design and implementation environment will enable effective benchmarking of the benefits of partial reconfiguration and high level synthesis.

Contents

Acronyms	v
List of Figures	vi
List of Tables	viii
1 Introduction	1
2 Background	5
2.1 Reconfigurable Computing	5
2.2 Design Methodologies	12
3 Approach	21
3.1 Models and Abstractions	22
3.1.1 Computation and Communication Models	23
3.1.2 Reconfiguration Model	26
3.1.3 Programming Model	28
3.2 Design Flow	29

3.2.1	Design Entry and Partitioning	31
3.2.2	Simulation	32
3.3	Implementation Flow	34
3.3.1	Reconfigurable Computing Specification Format	36
3.3.2	Configuration Management	37
4	Plan of Work and Preliminary Results	40
4.1	Plan of Work	40
4.2	Design Flow Progress	42
5	Conclusion	47
	Bibliography	49
A	Impulse C Sample Application	57

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

List of Figures

2.1	Typical FPGA structure.	6
2.2	Schaumont's taxonomy of reconfiguration (from [17]).	8
2.3	Virtual hardware for increasing an application's breadth (a) and depth (b).	10
3.1	Combined design and implementation flow.	22
3.2	Secure network streaming application.	24
3.3	Mutually exclusive set of processes.	27
3.4	SDR modes of operation.	28
3.5	Proposed design flow.	30
3.6	Architecture-specific implementation flow.	35
3.7	Xilinx Virtex configuration architecture.	38
4.1	Estimated project timeline.	42
4.2	DR Impulse C Configuration Function.	45
4.3	DR Impulse C reconfigurable set declaration.	45
4.4	DR Impulse C configuration control.	46

A.1	CSP structure for “Hello World!”	57
A.2	Impulse C configuration function for “Hello World!”	59
A.3	Impulse C process definition for software-implemented functions.	60
A.4	Impulse C process definition for the hardware-implemented function.	61
A.5	“HelloWorld” Impulse C simulation output.	61

List of Tables

2.1	Previous RTR Development Environments	20
-----	---	----

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 FPGA 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 [1] [2] [3]. 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 [4] [5] [6], 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 skeleton design flows 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) [7] is prompting FPGA vendors to provide backend tool support for RTR [8].

The objective of this project 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 will enable the use of commercial design entry and simulation tools, fully incorporate embedded processors into the computational and programming models, and permit partial reconfiguration of one or more FPGAs through automatic generation of custom configuration controllers.

The proposed research will offer the following contributions:

- Creation of a design and implementation flow for streaming RTR applications. The proposed flows provide 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 proposed design flow can utilize a variety of tools for design capture, as a test bed a commercial high-level synthesis tool will be 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, the proposed research will provide 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 proposed design flow and environment will be used to

implement encryption and image processing applications, domains well suited to the approach's computational model. These applications will 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 summarizes the progress made thus far on the high level design entry and simulation environment and presents a plan of work. Finally, conclusions are discussed in Chapter 5.

Chapter 2

Background

2.1 Reconfigurable Computing

Reconfigurable computing, 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 [9]. General purpose processors, such as are found in desktop computers, can be 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 [10] [11], none have attained commercial success. Given the prevalence of FPGAs in industry and academia, most research focuses on FPGAs, although many of the techniques and principles developed for FPGAs could be applied to other architectures as well.

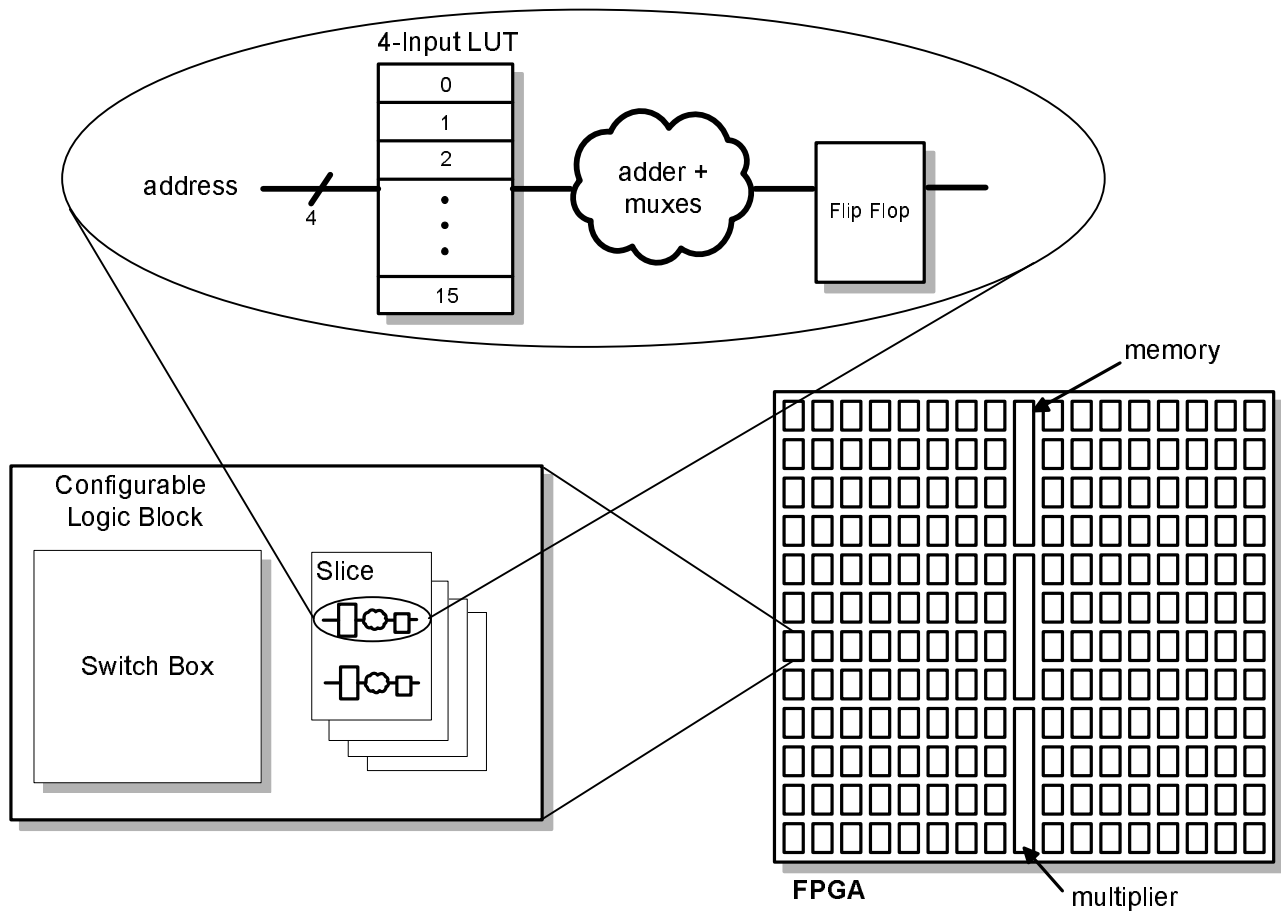


Figure 2.1: Typical FPGA structure.

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 processors 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 – 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) [12], cryptography [13], network applications [2], bio-informatics [14], and image processing [15].

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 [16] could be fixed in an FPGA with a corrected 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 Schau-mont [17] serves to illustrate and categorize the possible applications of RTR in hardware designs. Shown in Figure 2.2, this taxonomy maps the design space of RTR applications using three axes: architectural features, abstraction level, and configuration binding time.

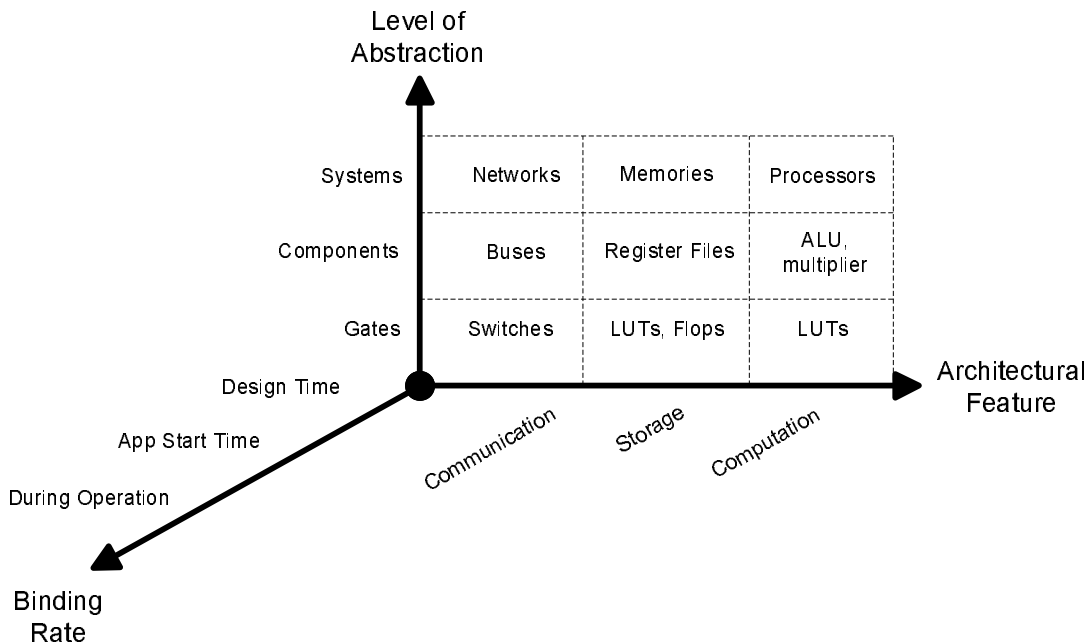


Figure 2.2: Schaumont’s taxonomy of reconfiguration (from [17]).

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.

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 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 breadth of a design, by swapping in and out functionality as required. Figure 2.3 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 [1], image and video processing [25], network applications [2], electronic design automation [26], neural networks [3], and instruction set extension [27]. 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

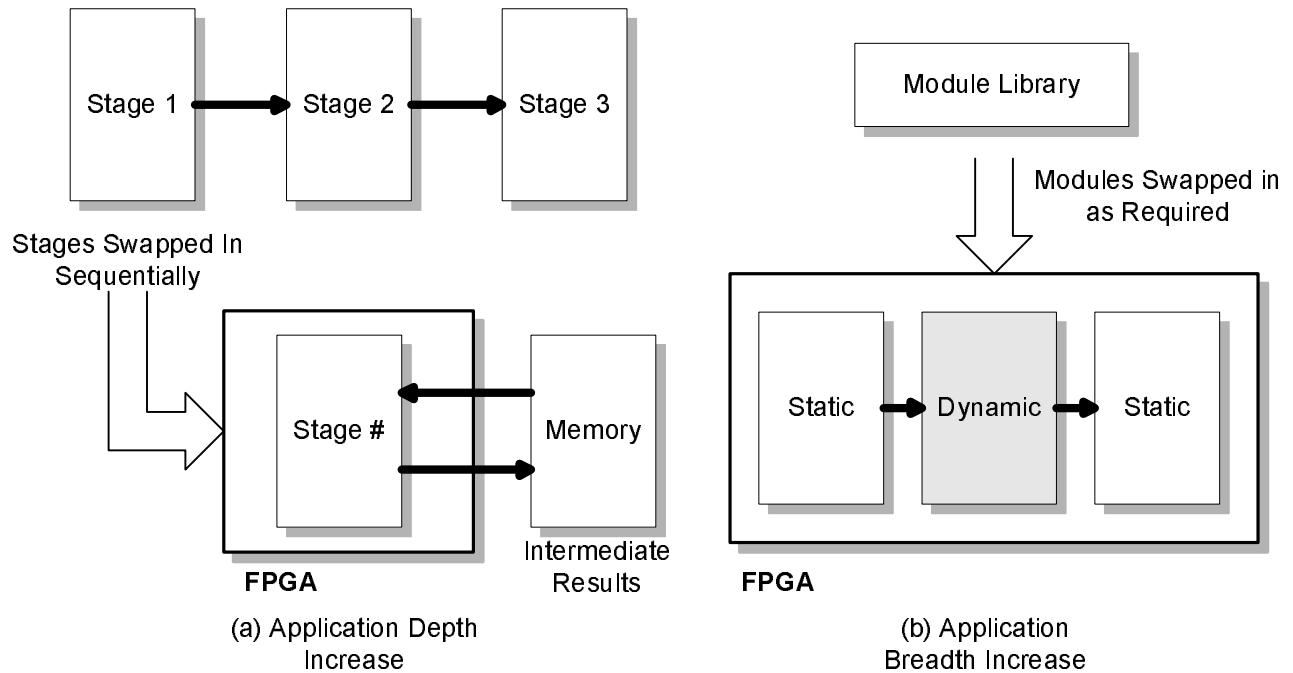


Figure 2.3: Virtual hardware for increasing an application's breadth (a) and depth (b).

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 [3] [29], where constant coefficient multipliers are updated at run-time, gene sequencing [1], 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 [1] 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 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. Furthermore, JBits forces the designer to work at a low level, complicating the design process. 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 Moores Law have doubled transistor density every 18 months. The first integrated circuit designs were completely hand crafted, limiting devices to hundreds 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. Many 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.

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 [44]. 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 [45], Impulse Accelerated Technologies' Impulse C [46], and Nallatech's DIME-C [47]. SRC Computer's CARTE design environment generates hardware from a C or FORTRAN description [48]. Model-based design tools, marketed toward DSP applications, are being offered by Xilinx with its System Generator [49].

HLS is not without its critics, who point out the suboptimal designs HLS produces compared to experienced hardware engineers. 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 [50].

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.
- Simulation of dynamic hardware. All commercial hardware simulators implicitly assume that the hardware is static.

- Design abstractions for reconfiguration. While synthesizable HDL may be easily ported from one configurable architecture to another, configuration interfaces vary across architectures, necessitating redesign unless suitable abstractions exist.

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 [4] 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 [51] at a high-level language, GDL, encountered difficulty in solving multiple NP-complete problems associated with HLL-to-gates synthesis. 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 [52] 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 [53] 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 [54] 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) [5] 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 Institute for Software Integrated Systems (ISIS) describes a prototype model-integrated design environment for dataflow applications [55]. 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 [56] 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 the proposed research. By defining abstractions of low-level details, a HLL-based approach to RTR application development is described [57]. 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 [45], 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 [58]. 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

Table 2.1: Previous RTR Development Environments

Chapter 3

Approach

The proposed research will develop a design and implementation flow that simplifies dynamic hardware application development through the use of high-level synthesis and abstractions of low-level details, addressing deficiencies in existing application development environments supporting dynamically reconfigurable hardware. A novel feature of the proposed work is the incorporation of partial reconfiguration, on-chip configuration management, and seamless integration of embedded processors.

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 architecture-specific backend flow, the output of which are the required configuration streams for the FPGA. The interface between the design and implementation flows is neutral from an architecture and design environment standpoint, permitting the free exchange and porting of the designs between architectures.

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

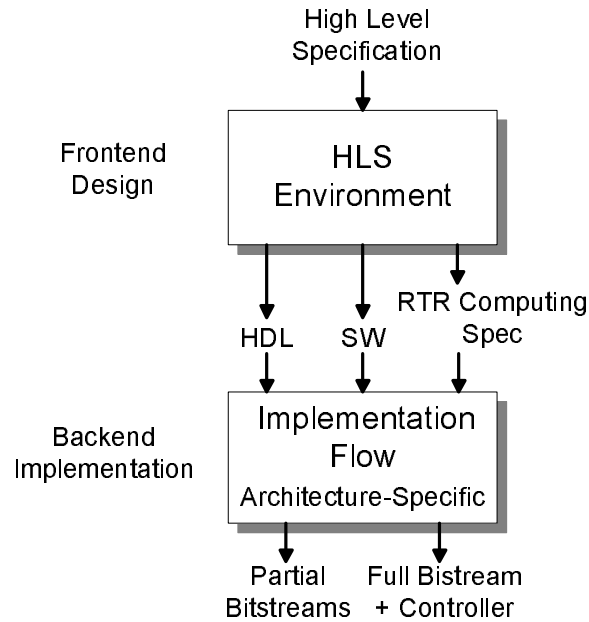


Figure 3.1: Combined design and implementation flow.

implementation flow.

3.1 Models and Abstractions

Models and abstractions are very important in hardware design, simplifying the design process by limiting the design space and hiding low-level details. 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 FSM, continuous time, discrete event, 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 the proposed research fully integrates embedded processors into RTR design, a programming model is required to interface these processors with the dynamic hardware.

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

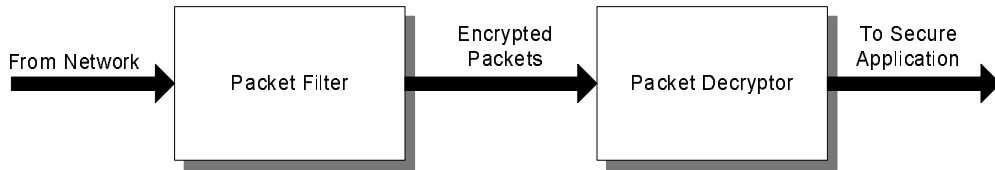


Figure 3.2: Secure network streaming application.

logic. These applications generally benefit from low-overhead, high bandwidth communication channels and deep computational pipelines. Streaming applications are generally decomposed into a pipeline of independent computational elements connected via unidirectional data streams. These computational elements function concurrently to one another, sharing no state or information except the data that is passed via the 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 [59].

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 [60]. A KPN is a collection of concurrently executing processes that communicate via in-

finitely 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 [61]. 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 [46], the Occam programming language [62], and the FDR2 refinement checker [63]. Additionally, with few exceptions, the many products that support dataflow models of computation such as KPN can be utilized within the CSP model. Such products include the commonly used graphical development environment of MathWorks' Simulink [64]. Traditional HDL design tools can also easily capture and implement streaming applications using the CSP model. For these reasons, CSP has been selected as the computational model for this project.

CSP, as originally conceived, includes operators to describe non-deterministic behavior. For the purposes of this project 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 per-

mits easy integration with Xilinx embedded processors as both the Xilinx MicroBlaze soft processor [65] and later versions of the PowerPC processor feature Fast Simplex Link (FSL) interfaces that are nothing more than asynchronous FIFO buffers.

The framework specifies the CSP model of computation and communication, first proposed by Hoare [61]. The application is divided into separate processes that run concurrently, similar to a multi-threaded computing environment. Unlike traditional multithreading, however, communication between processes occurs through message passing channels with no shared state. Processes block when reading from or writing to these data streams, forcing synchronization of processes.

The CSP model was selected for its suitability for describing streaming applications and its ease of implementation. A streaming application can be described using the CSP model by dividing the application into a series of sequential operations connected by simple communication channels. This high-level description permits the designer to extract a considerable amount of concurrency from the application, easing the work of the high-level synthesis tools.

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.

3.1.2 Reconfiguration Model

A methodology for configuration control has been selected that is centered on the idea of mutually exclusive modules and operating modes. 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 an image

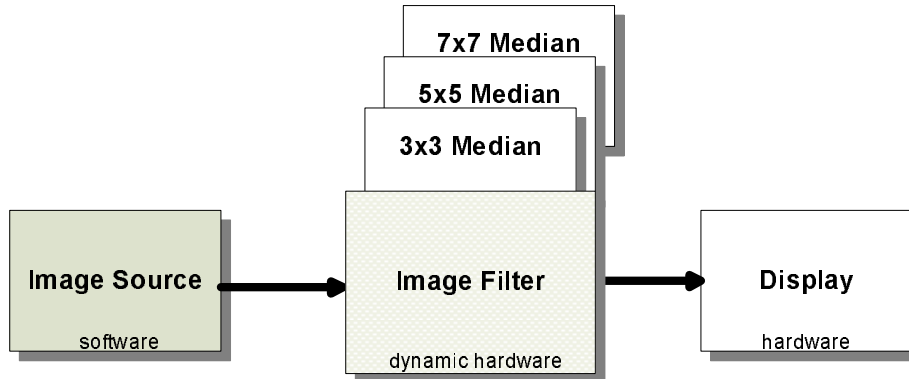


Figure 3.3: Mutually exclusive set of processes.

processing application. The user may want to filter the image with different filters. However, there is no need to implement in hardware every possible image filter the user may want to use. The figure shows a set of these mutually exclusive median image filters, 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 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].

For many RTR applications, the static nature of the communication connections described by the mutually exclusive set model is sufficient. For some applications, however, communication connections may need to be modified during operation. For example, consider an SDR that can switch between a transmit and receive mode. As shown in Figure 3.4 many modules are the same between the two modes but the connections between them are significantly different. The reconfiguration model permits the designer to specify different modes of operation. During operation, when a mode change occurs, the configuration manager determines which hardware and software modules from the current mode may be reused in the next mode. These modules will remain at their present location in hardware, reducing the reconfiguration time. The FIFO-based communication model reduces the importance of

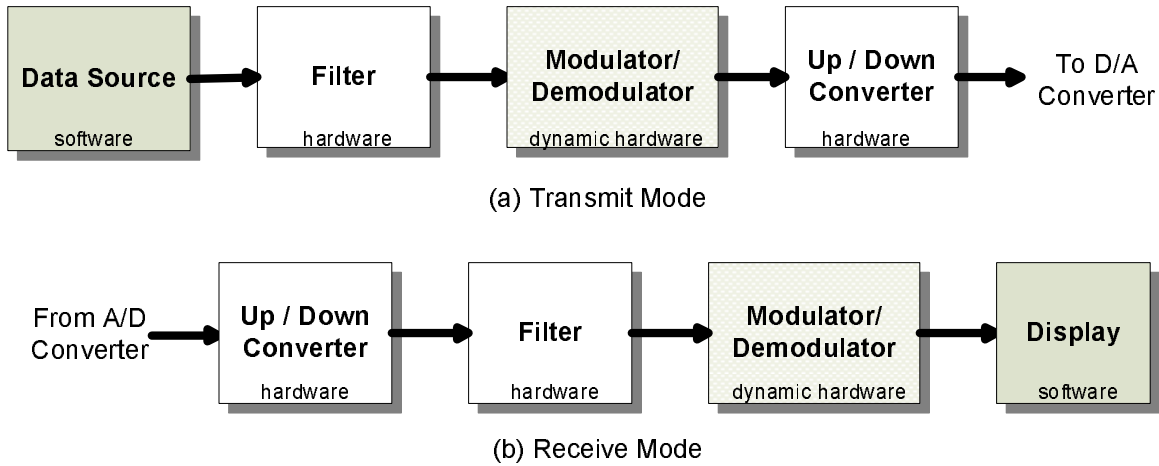


Figure 3.4: SDR modes of operation.

placement on communication throughput. Increased communication latency resulting from a poor placement of modules may be addressed by inserting additional storage elements in the communication streams. This permits the throughput to remain high by introducing additional cycles of latency.

This reconfiguration model permits 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 an integral part of many systems. For FPGA-based design 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.

This project’s programming model tightly integrates software with the dynamic hardware through a low-latency message passing interface. The programmer may directly communicate with the dynamic hardware via reads and writes to the FIFO-based data streams. This approach, adopted by Williams and Bergman [66] 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 project 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, both of which are supported by this project’s high-level development environment.

3.2 Design Flow

The proposed approach consists of a high-level design flow, specifying models of computation, communication, and control, while providing flexibility in each stage’s implementation. Based on the CSP model of computation [61] the methodology is targeted toward dataflow applications typically found in networking, signal processing, and cryptography domains. Communication is provided via synchronous unidirectional channels, called streams, which resemble FIFO queues. The system’s configuration is controlled from a dedicated configuration manager that abstracts the architecture-specific low-level complexities of RTR. A standard set of configuration functions is defined that may be invoked from within the application to request system modifications.

The proposed design flow, shown in Figure 3.5, 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

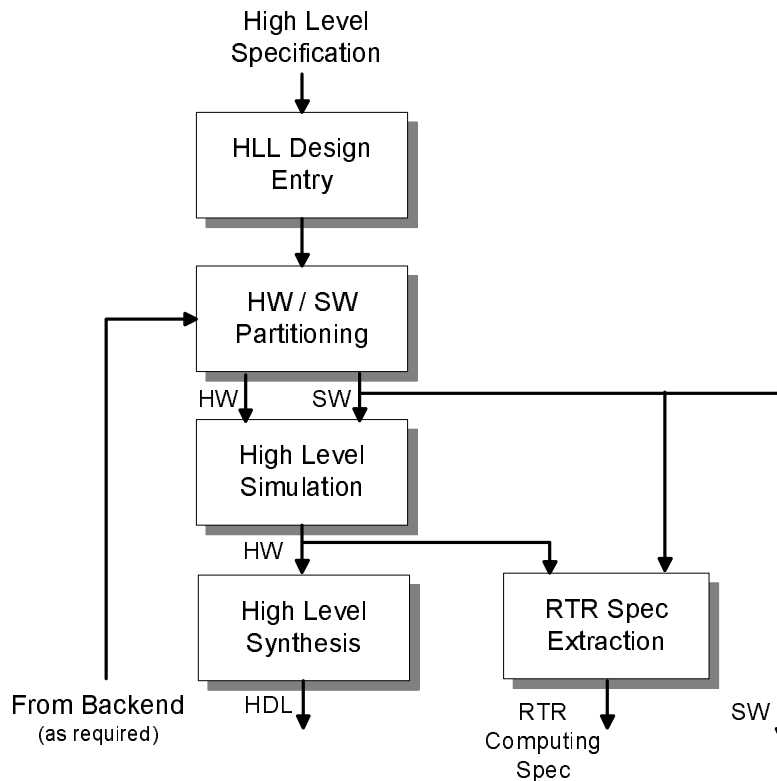


Figure 3.5: Proposed design flow.

model, produce synthesizable HDL for implementation by the architecture-specific backend flow. The inherent 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 Impulse C, may be used. While for embedded, timing-critical applications a library-based approach, such as found in Xilinx System Generator, may be appropriate.

While 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.

After partitioning, a high-level simulation, discussed below, is performed. The design is then compiled to RTL HDL by the high-level synthesis tools. Concurrent with this synthesis, the application description is analyzed to create a custom configuration manager. Owing to the architecture-dependent nature of partial reconfiguration, aspects of these tools must be targeted for a specific device family.

The final step in the frontend design flow is an optional HDL simulation, not shown in the figure. Functionality selected for software implementation can be simulated by instantiating an HDL model of the processor or through integration with an Instruction Set Simulator (ISS). This provides cycle-accurate verification of functionality. RTR is simulated by using multiplexers to select active RTR modules in a manner similar to Luk's [21]. The configuration manager controls the select line of the multiplexers. To permit cycle-accurate simulation of the reconfiguration process, the simulation model of the configuration controller optionally incorporates the actual configuration delays. This permits the calculation of application performance, in terms of clock cycles. Given the computational expense of simulating milliseconds of actual hardware time, it is envisioned that cycle-accurate reconfigurations would only be simulated if problems were discovered in the hardware implementation.

3.2.1 Design Entry and Partitioning

The proposed design flow makes use of commercial-quality high-level development tools for design entry. While the prototype implementation will utilize Impulse C [46], any high-level development environment supporting the CSP model may be used, including AccelDSP [67] and System Generator [49]. Regardless of the specific environment used, the same procedure is followed for design entry.

Design entry begins by partitioning the high-level specifications into separate modules. For the streaming applications targeted by this project 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 as well, a designer familiar with the application will likely be better at extracting high-level, coarse-grained parallelism than an algorithm. 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 partitioning chosen.

Hardware / Software partitioning is performed under direct designer control. In spite of years of research [40], no commercially successful automated partitioning tools exist. Additionally, 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 [4]. By simulating the entire design early in the design cycle functionality can be quickly verified and any integration issues identified. Furthermore, behavioral simulations run much faster than gate-level simulations, permitting more thorough tests to be run.

The proposed 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 thread communicating via blocking reads and writes to shared memory.

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.

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, a dynamic hardware design with a dedicated configuration controller. This controller will initiate reconfiguration if the system's output exceeds some threshold. As the controller must take some finite amount of time to determine if this threshold has been crossed, the system's output depends on when the controller decides that the threshold has been crossed. The results of a high-level simulation would likely differ from that of a hardware implementation.

The proposed research will investigate ways to reintroduce determinism into high-level dynamic hardware simulation. It is envisioned that the addition of a communication scheme that synchronizes reconfiguration to communication will provide the desired determinism.

However, for many applications the fast implementation time this approach provides may negate the requirement for deterministic high-level simulations as the design can simply be placed in hardware.

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 [68] 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 [8]. To mitigate the affects of modifications to the configuration architecture or development tools, this project 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 3.6, accepts HDL and software from the frontend design phase. Commercial synthesis tools convert the RTL HDL into a gate-level netlist. Based on the size of the resulting modules, the design is area constrained and, if a multi-FPGA system, partitioned across the devices using tools developed for this project.

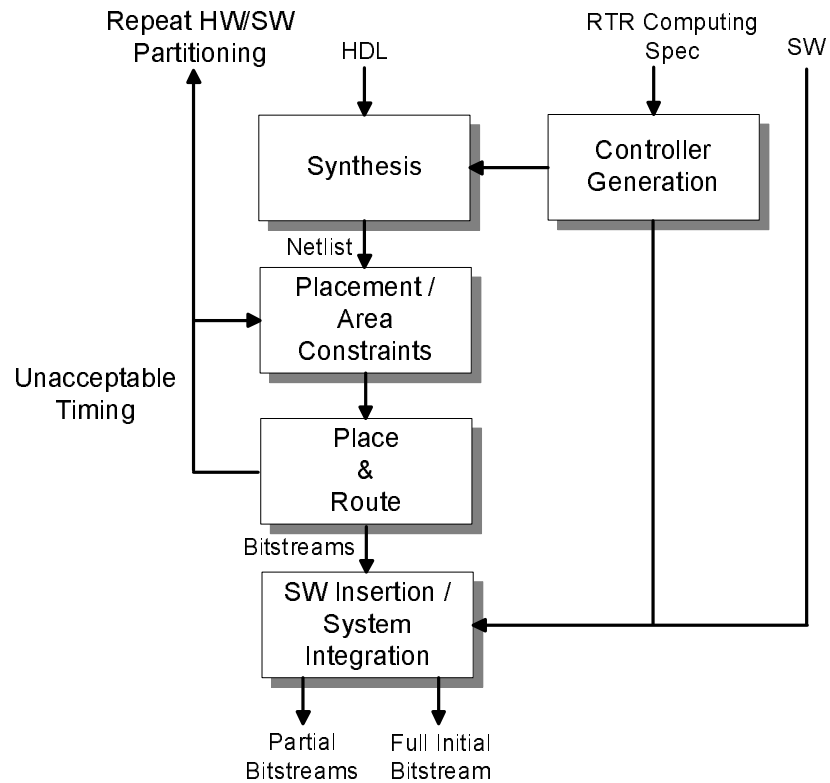


Figure 3.6: Architecture-specific implementation flow.

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.

3.3.1 Reconfigurable Computing Specification Format

In any domain a methodology is required for capturing the design. This methodology 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 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 [68] 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 [69] [70].

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 project will address the deficiencies in

existing work by developing a flexible file format, the RCSF. To facilitate acceptance of the format it is being developed with assistance from a related research project. The intention is to not tie the designer to a specific development environment or device architecture. Additionally, releasing the format to the public with implementation scripts and tools targeting the Xilinx partial reconfiguration flow should improve acceptance.

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, and a link to its compiler.
- List of external resources required (memories, I/O, etc.)
- Architecture-independent configuration control information.

Much effort will go into capturing the configuration manager, as this is an important design component that has been neglected by others. Depending on the method of configuration management, this may be as simple as providing a listing to software that handles reconfiguration or as involved as specifying the FSM for the controller.

3.3.2 Configuration Management

The configuration architecture of Xilinx FPGAs is shown in Figure 3.7. The device is configured by the loading of configuration data in segments called frames. A frame runs

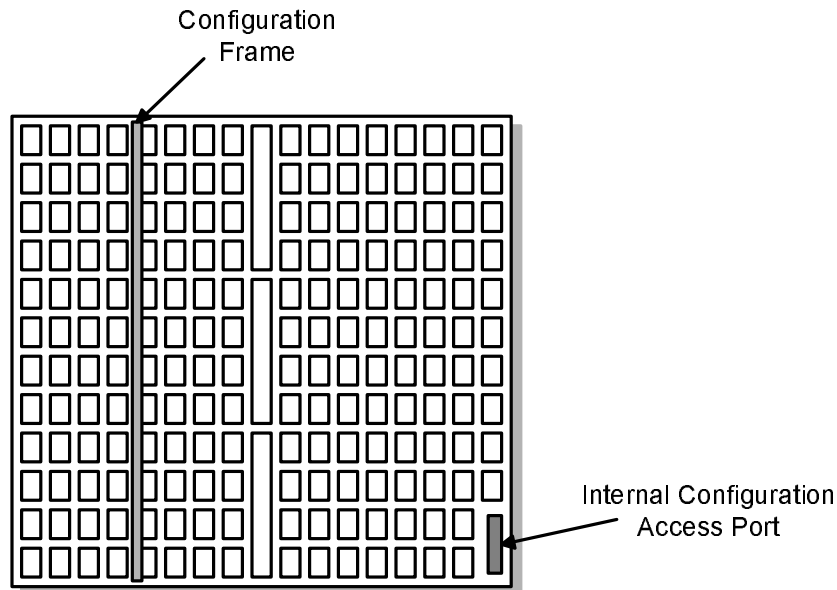


Figure 3.7: Xilinx Virtex configuration architecture.

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.

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 ICAP 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 more ambitious reconfigurations this controller may need to perform bitstream manipulations to reposition existing partial

bitstreams or to dynamically modify inter-module connections.

For simple RTR applications this configuration management may be performed by a FSM. For more complex designs it 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. A key contribution of this project is the development of a design specification methodology that is independent of the FPGA family or design capture environment used.

Chapter 4

Plan of Work and Preliminary Results

4.1 Plan of Work

This project will leverage existing partial reconfiguration implementation tools and techniques developed by other projects in Virginia Tech's Configurable Computing Machine (CCM) Laboratory whenever possible. Collaboration with several of the CCM Lab's projects will be explored, particularly in defining the RCSF. These interactions will increase the relevance and visibility of this research.

As described in the following section, initial simulation and design capabilities have been added to a high-level development environment. The remaining tasks in the proposed research include:

- Design Flow
 - *Simulation Libraries* Additional functionality is required to support the operating mode abstraction. This involves further modifications to the Impulse C simulation libraries.
 - *Controller Specification* The methods for specifying the configuration manager

must be codified.

- *RCSF* The exact format of the RCSF will be finalized with input from other related projects. Support scripts must be created to parse Impulse C code to generate the RCSF and perform HLS.

- Implementation Flow

- *Implementation Constraint Generation* FPGA implementation flows require placement constraints for reconfigurable regions and timing constraints for all logic. Automated tools will be developed to examine the HDL produced from the Impulse C synthesizer and apply appropriate constraints. Existing work in area estimation and optimum placement will be leveraged.
- *Configuration Manager* Utilizing existing ICAP controllers, if possible, the configuration manager will be automatically created from the RCSF specification. In the simplest case this involves the automated instantiation of an embedded processor connected to external memory and the ICAP. Additional supported configuration schemes will involve the generation of an FSM in HDL to implement the controller.
- *System Integration* Once created, the partial bitstreams must be stored externally to the FPGA and their location must be made known to the configuration manager. Furthermore, the initial bitstream and partial bitstreams must be packaged in a loadable format.

- Benchmark Applications

- *Benchmark Creation* Through coordination with related projects, a set of benchmarks will be identified to quantify performance, taking into account reconfiguration time, throughput, and development time.
- *Application Development* Using the identified benchmarks, applications will be created to quantify the performance benefit of partial RTR and the development

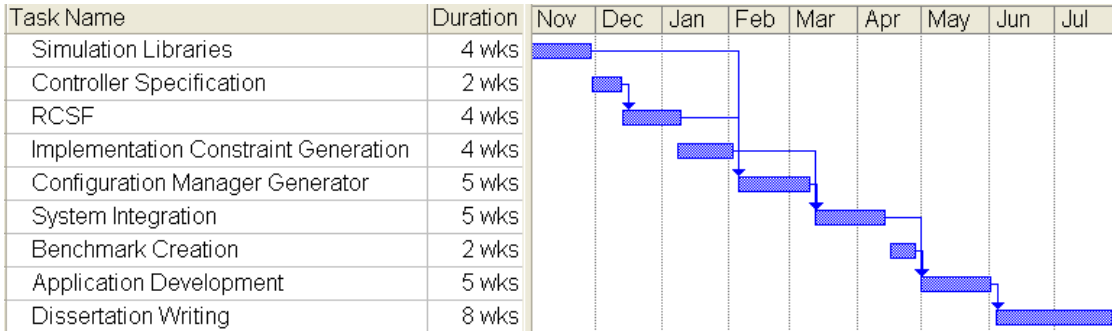


Figure 4.1: Estimated project timeline.

time reduction of HLS.

Several of the required tasks related to partial bitstream creation may benefit from collaboration with other projects in the CCM Lab, complicating the creation of an accurate project schedule. Figure 4.1 presents an aggressive timeline, completing the work within seven months with an additional two months spent on the dissertation. Unforeseen issues and opportunities may likely extend this by several months. Nonetheless, the figure demonstrates a rough estimate of the relative effort required for each task.

4.2 Design Flow Progress

Impulse C [46] is an ANSI C-based language utilizing the same stream and process abstractions as Los Alamos National Lab’s Streams-C work [43]. Utilizing the CSP model, Impulse C permits the application developer to describe hardware using a large subset of standard C. The developer first decomposes the application into concurrent processes connected by high-throughput low-latency streams implemented as FIFO connections. In Impulse C these processes are defined as standard C functions accepting a special stream data type as arguments. When compiling the code for high-level simulation, each process becomes a concurrently running thread with synchronization occurring through blocking reads and writes to the connecting streams. Processes selected for software implementation may utilize all ANSI

C functionality while those marked for hardware must obey certain language limitations. An example program is present in the Appendix.

While the Impulse C simulation and implementation 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 Impulse C language becomes a powerful development framework for dynamic reconfiguration of FPGA hardware.

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 libraries. Preliminary modifications to the simulation libraries have been performed that permit dynamic hardware to be simulated at a high level using the Impulse C design tools. This modified language is referred to as DR Impulse C, highlighting its Dynamic Reconfiguration (DR) ability.

The 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 Impulse C `co_streamread` and `co_streamwrite` 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. Otherwise, communication proceeds as normal.

To describe RTR applications in DR Impulse C, the programmer defines sets of mutually exclusive Impulse C processes. At run-time, a configuration controller process can reconfigure

the hardware, swapping one process in the reconfigurable set for another. New functions create these reconfiguration sets (`co_reconfig_create`) and select a new dynamic process to execute in hardware (`co_architecture_config`).

The following steps provide an overview of the application description process by expanding the “HelloWorld” example from the Appendix (see Figure A.1) to include dynamic hardware modules. Two dynamic modules exist in this example. The default DoText module merely acts as a pass through, streaming the input characters out without modification. DoText may be dynamically replaced by a case-swapping ModText module.

Step 1) Create processes.

Process declaration and creation occur in the architecture’s configuration function as before, with the exception that multiple processes can source and sink the same streams as long as they are mutually exclusive (i.e. only one process driving a stream is active in hardware at a time). Figure 4.2, below, presents the configuration function for this application.

Step 2) Assign processes to reconfigurable sets or hardware.

Processes created in Step 1 can be implemented in static hardware, dynamic hardware, or in software on an embedded microcontroller. Static hardware implementation is indicated by placing a `co_loc` attribute on the process through the `co_process_config` function. Dynamic hardware implementation is specified by passing the process as an argument to the `co_reconfig_create` function. Processes not assigned to hardware are implemented in software in an embedded processor. The configuration function code required to define a mutually exclusive set of processes is presented in Figure 4.3, below.

Step 3) Create DR controller.

Dynamic reconfiguration must be controlled by a single process in simulation. The new function `co_architecture_reconfig` initiates reconfiguration, as shown in Figure 4.4, blocking until completion. Behind the scenes, `co_architecture_reconfig` loads the selected

```

1  void config>HelloWorldReconfig(void *arg)
2  {
3      :
4      DoText_process = co_process_create("DoText", (co_function)DoText,
5                                          2,
6                                          StreamA,
7                                          StreamB);
8
9      producer_process = co_process_create("Producer", (co_function)Producer,
10                                         1,
11                                         StreamA);
12
13     ModText_process = co_process_create("ModTextA", (co_function)ModText,
14                                         2,
15                                         StreamA,
16                                         StreamB);
17
18     consumer_process = co_process_create("Consumer", (co_function)Consumer,
19                                         1,
20                                         StreamB);
21     :
22     :

```

Figure 4.2: DR Impulse C Configuration Function.

```

1  void config>HelloWorldReconfig(void *arg)
2  {
3      :
4      // Declare reconfigurable set of modules
5      co_reconfig Set;
6
7      // Define which processes are dynamically swappable
8      // First process listed is default process
9      Set = co_reconfig_create("test reconfig set", 2, DoText_process, ModText_process);

```

Figure 4.3: DR Impulse C reconfigurable set declaration.

```
1 | void Producer(co_stream StreamA)
2 | {
3 |     :
4 |     // Reconfigure hardware to use ModText in place of default DoText
   |     co_architecture_reconfig("test reconfig set A", "ModTextA");
```

Figure 4.4: DR Impulse C configuration control.

process's partial bitstream from memory, streaming it to the ICAP.

Chapter 5

Conclusion

Configurable logic devices are routinely used to create custom computational hardware, improving performance beyond that of processors without requiring the lengthy design, manufacturing, and testing cycle of a dedicated integrated circuit. A domain that frequently employs configurable logic devices, and FPGAs in particular, is that of streaming, or dataflow, applications, such as found in DSP, cryptography, or network applications. While impressive results are obtained by treating an FPGA as static hardware, multiple projects have demonstrated additional performance enhancements when the hardware is dynamically modified during operation. RTR permits a small device to emulate a much larger one, with the active logic tailored to the conditions at hand.

In spite of the great potential of RTR, existing design methods require significant low-level and manual work, severely hindering the utility and thus the adoption of this technology. Several projects have addressed the issue, describing environments that capture design intent at a much higher level. Recent advances in configurable computing have highlighted omissions in these previous works, including obsolete design entry methods, lack of comprehensive abstractions and models, inability to describe partial reconfiguration, and requirement of a dedicated external host.

Capitalizing on recent FPGA architectural advances, the proposed research defines 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.

The proposed research will create an end-to-end design and implementation environment, permitting, for the first time, the high-level development of autonomous, partially reconfigurable applications. The applications created using this environment will serve as much needed benchmarks, quantifying the benefits of partial reconfiguration on modern architectures and the development time reductions of HLS. Furthermore, the open nature of the design and implementation flows will empower other researchers to extend this work to encompass additional high-level design capture environments, FPGA architectures, and reconfiguration strategies.

Bibliography

- [1] 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.
- [2] 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.
- [3] P. James-Roxby and B. Blodget, “Adapting constant multipliers in a neural network implementation,” in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), p. 335, IEEE Computer Society, 2000.
- [4] D. I. Lehn, R. D. Hudson, and P. M. Athanas, “Framework for architecture-independent run-time reconfigurable applications,” vol. 4212, pp. 162–172, SPIE, 2000.
- [5] I. Ouais, 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.
- [6] 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.

- [7] A. Rudra, “The rising importance of fpga technology in software defined radio,” *COTS Journal*, January 2005. January 2005.
- [8] A. Malagamba, “SDR prêt-à-porter,” *FPGA and Structured ASIC Journal*. February 28th, 2006.
- [9] J. Villasenor and W. Mangione-Smith, “Configurable computing,” *Scientific American*, pp. 66–71, June, 1997 June, 1997.
- [10] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, “A quick safari through the reconfiguration jungle,” in *Design Automation Conference*, pp. 172–177, 2001.
- [11] 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. Overview of everything reconfigurable.
- [12] 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.
- [13] 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.
- [14] 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*, 2005.
- [15] P. Athanas and A. Abbott, “Real-time image processing on a custom computing platform,” *Computer*, vol. 28, no. 2, pp. 16–25, 1995.
- [16] 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.

- [17] 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.
- [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 *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 *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 *2nd Annual Military and Aerospace Applications of Programmable Logic Devices Conference*, (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 *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 196–204, IEEE Computer Society Press, 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 *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 Tech, 2004.
- [31] B. Blodget, S. McMillan, and P. Lysaght, “A lightweight approach for embedded re-configuration 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 back-propagation algorithm using reconfigurable fpgas,” vol. 4, pp. 2097–2102 vol.4, 1994.
- [33] J. Hadley and B. Hutchings, “Design methodologies for partially reconfigured systems,” in *IEEE Symposium on FPGAs for Custom Computing Machines* (P. Athanas and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 78–84, IEEE Computer Society Press, 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 31st, 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 *processing of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 49–56, 2000.
- [44] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, "Automatic mapping of khoros-based applications to adaptive computing systems," in *Military and Aerospace Applications of Programmable Devices*, (Washington, D.C.), 1999.
- [45] Celoxica, Inc., "Handel-c for hardware design," white paper, 2006.

- [46] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Upper Saddle River, N.J.: Prentice Hall, 2005.
- [47] Nallatech, “Dimetalk v3.0 application development environment,” product brief, 2006.
- [48] D. Poznanovic, “Application development on the src computers, inc. systems,” in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.
- [49] Xilinx, Inc., “Xilinx system generator for DSP version 8.2,” user’s guide, 2006.
- [50] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, and A. Troxel, I. George, “Survey of c-based application mapping tools for reconfigurable computing,” in *International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, (Washington, D.C.), 2004.
- [51] 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.
- [52] M. Eisenring and M. Platzner, “A framework for run-time reconfigurable systems,” *J. Supercomput.*, vol. 21, no. 2, pp. 145–159, 2002.
- [53] 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.
- [54] 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.

- [55] 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.
- [56] W. Luk, N. Shirazi, and P. Y. K. Cheung, “Compilation tools for run-time reconfigurable designs,” in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, p. 56, 1997.
- [57] 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.
- [58] A. L. Slade, B. E. Nelson, and B. L. Hutchings, “Reconfigurable computing application frameworks,” in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pp. 251–260, 2003.
- [59] S. Craven and P. Athanas, “Examining the viability of fpga supercomputing,” 2006. Submitted to EURASIP Journal on Embedded Systems.
- [60] E. A. Lee and T. M. Parks, “Dataflow process networks,” in *Proceedings of the IEEE*, pp. 773–799, May 1995.
- [61] C. Hoare, “Communicating sequential processes,” *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [62] P. Welch and D. Wood, “The Kent Retargetable occam Compiler,” in *Parallel Processing Developments, Proceedings of WoTUG 19* (B. O’Neill, ed.), vol. 47 of *Concurrent Systems Engineering*, (Amsterdam, The Netherlands), pp. 143–166, World occam and Transputer User Group, IOS Press, Mar. 1996. ISBN: 90-5199-261-0.
- [63] Formal Systems, “Failures-divergence refinement,” user manual.
- [64] The MathWorks, “Simulink 6,” product brief.

- [65] Xilinx, Inc., “Microblaze processor reference guide,” reference manual, 2006.
- [66] J. A. Williams, N. W. Bergmann, and X. Xie, “Fifo communication models in operating systems for reconfigurable computing,” in *Field-Programmable Custom Computing Machines. 13th Annual IEEE Symposium on*, pp. 277–278, 2005.
- [67] Xilinx, Inc., “AccelDSP synthesis tool,” product information.
- [68] Xilinx, “XAPP290: Two flows for partial reconfiguration: Module based or difference based,” 2004.
- [69] 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.
- [70] C. Bobda and B. Ali Ahmadiania, “Dynamic interconnection of reconfigurable modules on reconfigurable devices,” *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 443–451, 2005.

Appendix A

Impulse C Sample Application

While a simple Hello World application can be represented in a variety of ways using Impulse C, Figure A.1 below illustrates one such implementation. Producer is a process that streams the ASCII characters of “HelloWorld!” to the DoText process. In this example DoText merely acts as a pass-through, streaming the characters out to the Consumer process for display.

The Impulse C code describing this application consists of three process functions (Producer, DoText, and Consumer), a main function, and two configuration functions. The main function, presented along with the configuration functions in Figure A.2, first initial-

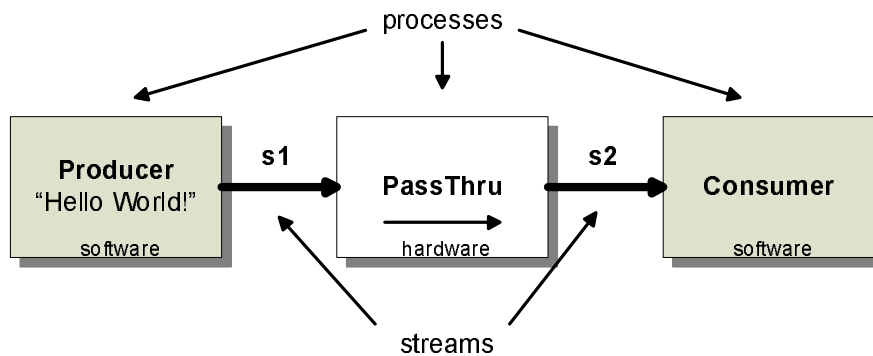


Figure A.1: CSP structure for “Hello World!”.

izes the architecture with a call to `co_initialize` before starting simulation with a call to `co_execute`. The `co_initialize` function calls an architecture creation function that is passed the name of the users configuration function, a special function describes the processes and their connections. These functions are only used for simulation and architecture creation and are not represented in an implemented design.

The functionality of each process is captured by standard ANSI C functions accepting Impulse C's stream data type as arguments, illustrated by the HelloWorld example's process functions in Figure A.3, for software implemented processes, and in Figure A.4, for hardware implemented processes. Calls to `co_streamread` and `co_streamwrite` read from or write to a stream's FIFO buffer, respectively. These are, by default, blocking calls in sticking with the CSP computational model. While additional communication mechanisms are available (non-blocking stream communication, registers, shared memory, and signals), streaming applications described by Impulse C heavily utilize blocking stream communication. For hardware-implemented processes there are some C language restrictions - pointers must resolve to addresses at compile-time, functions must be able to be in-lined, and shifting is limited to constant shift values.

Comparing Figures A.3 and A.4 show little difference between the code written for software implementation and that written for hardware implementation.

When the code in Figures A.2 through A.4 is compiled with the required libraries and executed, the three processes begin concurrent execution as Windows threads. The result of program execution is shown below in Figure A.5 for the first iteration.

```

1 // Main function initializes architecture, starts simulation
2 int main(int argc, char *argv[]) {
3     int iterations = 10; // Number of "HelloWorld" to send
4     co_architecture my_arch;
5
6     printf("Copyright 2003 Impulse Accelerated Technology, Inc.\n");
7     // Initialize architecture by calling function defined below.
8     my_arch = co_initialize(iterations);
9     // Begin execution of architecture.
10    co_execute(my_arch);
11    printf("Application HelloWorld complete.\n");
12    return(0);
13 }
14
15 // Create architecture using users configuration function
16 co_architecture co_initialize(int iterations)
17 {
18     return(co_architecture_create("HelloWorldArch", "generic", config_helloworld,
19         (void *)iterations));
20 }
21
22 // User-defined architecture configuration function
23 void config_helloworld(void *arg)
24 {
25     int iterations = (int) arg;
26     // Declare streams to connect processes
27     co_stream s1,s2;
28     co_process producer, consumer;
29     co_process dotext;
30
31     // Create streams, define FIFO buffer width (8) and size (2)
32     s1=co_stream_create("Stream1", 8, 2);
33     s2=co_stream_create("Stream2", 8, 2);
34     // Create processes and connect them to streams.
35     // producer is defined by the C function Producer and has 2 arguments: a stream (s1) and t
36     producer=co_process_create("Producer", (co_function)Producer, 2, s1, iterations);
37     dotext=co_process_create("DoText", (co_function) DoText, 2, s1, s2);
38     consumer=co_process_create("Consumer", (co_function) Consumer, 1, s2);
39     // Mark process dotext for hardware implementation in PE0
40     // Other processes execute in software by default
41     co_process_config(dotext, co_loc, "PE0");
42 }

```

Figure A.2: Impulse C configuration function for “Hello World!”.

```

1 void Producer(co_stream output_stream, co_parameter iparam)
2 {
3     int iterations=(int)iparam;
4     int32 i;    // Specify 32 bit integer
5     static char HelloWorldString[] = "HelloWorld!";
6     char *p;
7
8     // Open stream for writing data of size 8-bits
9     co_stream_open(output_stream, O_WRONLY, 8);
10    // Loop over string per requested # of iterations
11    for(i=0; i<iterations; i++) {
12        p = HelloWorldString;
13        // Write string out character-by-character
14        while (*p) {
15            co_stream_write(output_stream, p, sizeof(char));
16            p++;
17        }
18    }
19    co_stream_close(output_stream);
20 }
21
22 void Consumer(co_stream input_stream)
23 {
24     char c;
25     // Open stream for reading data of size 8-bits
26     co_stream_open(input_stream, O_RDONLY, 8);
27     // While stream is open, read from it and display data
28     while ( co_stream_read(input_stream, &c,
29         sizeof(char)) == co_err_none ) {
30         printf("Consumer read %c from stream: input_stream\n", c);
31     }
32     co_stream_close(input_stream);
33 }

```

Figure A.3: Impulse C process definition for software-implemented functions.

```

1 void DoText(co_stream input_stream, co_stream output_stream)
2 {
3     char c;
4
5     // Forever loop
6     do {
7         // Open streams
8         co_stream_open(input_stream, O_RDONLY, 8);
9         co_stream_open(output_stream, O_WRONLY, 8);
10        while ( co_stream_read(input_stream, &c, sizeof(char)) == co_err_none ) {
11 // Pragma tells CoDeveloper synthesis tool to pipeline this loop
12 #pragma CO PIPELINE
13         // Text processing would occur here
14         co_stream_write(output_stream,&c,sizeof(char));
15     }
16     co_stream_close(input_stream);
17     co_stream_close(output_stream);
18 } while (1);
19 }

```

Figure A.4: Impulse C process definition for the hardware-implemented function.

```

1 Copyright 2003 Impulse Accelerated Technology, Inc.
2 Consumer read H from stream: input_stream
3 Consumer read e from stream: input_stream
4 Consumer read l from stream: input_stream
5 Consumer read l from stream: input_stream
6 Consumer read o from stream: input_stream
7 Consumer read W from stream: input_stream
8 Consumer read o from stream: input_stream
9 Consumer read r from stream: input_stream
10 Consumer read l from stream: input_stream
11 Consumer read d from stream: input_stream
12 Consumer read ! from stream: input_stream

```

Figure A.5: “HelloWorld” Impulse C simulation output.