

# **A High-Level Development Framework for Run-Time Reconfigurable Applications**

Stephen Craven and Peter Athanas  
Bradley Department of Electrical and Computer Engineering  
Virginia Polytechnic Institute and State University, Blacksburg, VA 24061  
{scraven | athanas@vt.edu}

## **Abstract**

Dynamically modifying computational hardware during operation has long held great promise for increasing the performance and reducing cost. However, designing Run-Time Reconfigurable (RTR) applications has been prohibitively difficult. Recent advances in FPGA architectures and implementation tools better support dynamic hardware development. This project presents the foundation of an integrated development environment for dynamic hardware applications that raises the level of abstraction from the gate-level to a high-level specification. By extending a commercial high-level synthesis tool, the presented methodology permits RTR applications to be created directly from a high-level description.

## **I. Introduction**

Configurable computing devices, such as Field Programmable Gates Arrays (FPGAs), permit arbitrary reprogramming of the devices' hardware structure after manufacture. This programmability greatly decreases development costs and time-to-market compared to Application-Specific Integrated Circuits (ASICs). FPGAs are routinely used for Digital Signal Processing (DSP), cryptography, bio-informatics, and network applications. These streaming application domains consist of a repeatable schedule of computations operating on a steady flow of data. These applications generally benefit from low-overhead, high bandwidth communication channels and deep computational pipelines, all strong suits of configurable logic.

Almost all commercial designs using FPGAs treat the configurable logic as static in the deployed product. However, previous research has demonstrated performance benefits from partially or fully reconfiguring an FPGA during operation [1]. By tailoring the circuit dynamically to the problem at hand, significant performance gains are possible. Replacing unused circuits with additional functionality at run-time permits a smaller and cheaper device to be deployed.

Unfortunately, developing RTR applications has been a difficult undertaking. FPGA design tools share a lineage with ASIC tools, with both assuming static designs. What little vendor support that has been available has forced the designer to do much manual, low-level work. Commercial tools in the form of design capture methodologies and simulators are non-existent. An FPGA designer developing a RTR application is very much a trailblazer, forced to improvise the required tools with little assurance that the final design will even function.

Modern digital designs, whether they target ASICs or FPGAs, are complex. Shrinking transistors prompt designers to include more functionality inside a single component. The most timing-consuming design steps currently occur at the Register Transfer Level (RTL), with designers manually describing the flow of data between registers. To increase designer productivity some in research and industry are pushing for automating RTL design and other

time-consuming steps. This focus towards Electronic System Level (ESL) design is centered on tools that completely or partially automate the translation of a high-level specification or functional description into a Hardware Description Language (HDL) format ready for implementation. Several companies currently market tools supporting ESL design [2]. Many take a functional description in the form of a High Level Language (HLL), such as C or MATLAB, and synthesize a gate-level netlist. High Level Synthesis (HLS) tools, sometimes referred to as C-to-gates compilers, do not produce results that are performance comparable to hardware designed by an experienced engineer. However, much like C software compilers or automated place-and-route tools, HLS inefficiencies may be acceptable for many applications in light of the reduced design time.

As RTR hardware designs are among the most complex possible on FPGAs, owing to the myriad of low-level details that must be attended to, several researchers over the past decade have proposed development environments utilizing some form of HLS to automate RTR design. 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. While no means an exhaustive survey, some of the more prominent projects are described below.

Janus [3] was an early effort at a RTR application development environment centered on Java. Software for the host PC was written in Java while the hardware was created from JHDL, a Java-based structural hardware description language. A configuration controller residing on the host PC was automatically generated, managing the configurations of each FPGA in the system. Partial reconfiguration and dynamic scheduling were not supported.

The Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems (SPARCS) project [4] started with a behavioral VHDL description of the application separated into tasks communicating through shared memory or direct connections. Temporal and spatial scheduling occurred across multiple FPGAs. A high-level synthesis tool converted the behavioral description to RTL that was then processed with traditional tools.

The Institute for Software Integrated Systems (ISIS) describes a prototype model-integrated design environment for dataflow applications [5]. ISIS focused on constraint-driven development and verification. Tools automatically applied user-specified constraints to prune the design space. A complete runtime environment was described linking the dynamic hardware modules to a software OS. Design entry occurred via graphical tools linking to pre-described IP.

Gehring and H.-M.-Ludwig [6] described a framework of tools supporting RTR for the Xilinx XC6200 FPGA. The unique architecture of the obsolete XC6200 limits the applicability of their work to modern FPGAs.

Eisenring and Platzner's RTR Framework [7] described a tool-independent design and implementation methodology. The design was specified by a problem graph, an architecture graph, and a mapping between the two. This formalism simplified tool development. Hierarchical configuration control was achieved through a separate configurator node running on the host processor.

These previous projects all suffer from the omission of partial reconfiguration support. Additionally, most assume a model of external configuration control, mandating the use of a host

processor. For embedded applications this requirement may be prohibitive. The methods of high-level design entry are limited, in some cases requiring the designer to manually partition the design into a data flow graph or to assemble the application out of a pre-compiled library of hardware components. In the case of Janus the use of JHDL to describe hardware forces the designer to work at structural level. It is also interesting to note that no project has been extended since its initial implementation. This is perhaps in part due to the tight coupling of many of these frameworks to a specific architecture or esoteric design capture tool.

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 true two-dimensional reconfiguration is now available in some modern devices. The inclusion of embedded processors has freed dynamic hardware designs from their previous reliance on a host processor for configuration management, opening new application domains. The growing interest in software defined radios, with their dynamic creation of radio waveforms, is prompting FPGA vendors to provide backend tool support for RTR.

The objective of the work presented here is to define an architecture-agnostic RTR application development methodology, incorporating the latest advances and trends in configurable computing, to permit the rapid creation of dynamic hardware from a high-level specification. Previous research attempts, unaccepted even in their own time, are unable to capitalize on these current advances. Addressing the deficiencies in previous research, the presented methodology enables the use of commercial design entry and simulation tools, fully incorporates embedded processors into the computational and programming models, and permits partial reconfiguration of one or more FPGAs through automatic generation of custom configuration controllers.

## **II. Design and Implementation Methodology**

The methodology, shown in Figure 1, consists of a frontend design flow feeding a backend implementation flow. Separating the architecture-specific backend flow from the design and simulation frontend facilitates the porting of designs across different architectures and frees the methodology from a reliance on a single design entry environment. The HLS environment produces an HDL description of the hardware, along with software code describing any embedded programs. A new specification format is used by this project to capture the RTR-specific design attributes that cannot be easily described in an HDL, including the configuration management control logic.

### **A. Computational and Communication Models**

Along with the design and implementation flows, this methodology provides abstractions for communication, computation, and reconfiguration. The models of computation and communication were selected to favor the traditional applications of FPGAs, namely streaming applications.

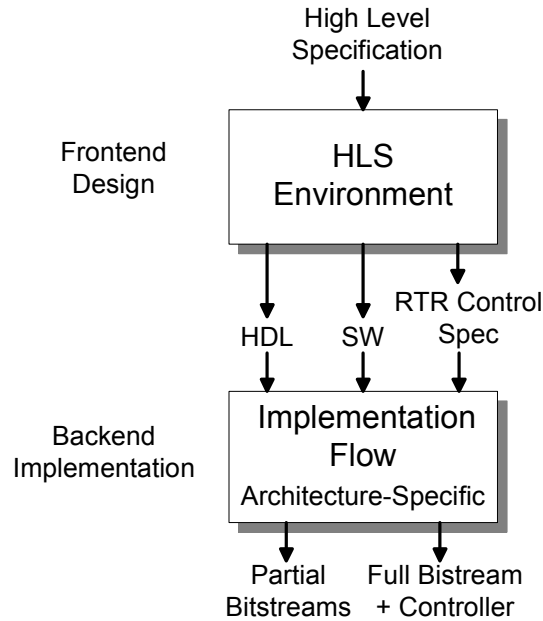


Figure 1. Combined Design and Implementation Flows.

The framework specifies the Communicating Sequential Processes (CSP) model of computation and communication, first proposed by Hoare [8]. 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 FIFO-based message passing channels with no shared state. Processes block when reading from or writing to these data streams, forcing synchronization.

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 FIFO-based communication permits easy integration with Xilinx embedded processors as both the Xilinx MicroBlaze soft processor [9] and later versions of the PowerPC processors embedded in some Xilinx FPGAs feature fast simplex link interfaces that are nothing more than asynchronous FIFO buffers.

## B. Frontend Design Flow

In following the design flow, shown in Figure 2, the designer describes the application using an HLL-based design entry tool. This design flow makes use of commercial-quality high-level development tools for design entry. While the prototype implementation utilizes Impulse C [10], any high-level development environment supporting the CSP model may be used. Regardless of the specific environment used, the same procedure is followed for design entry.

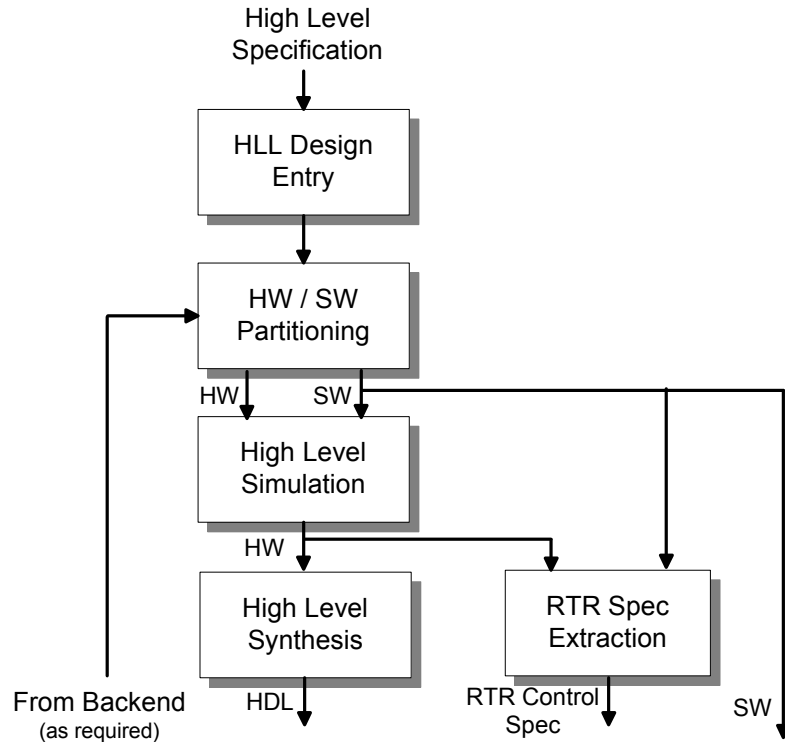


Figure 2. Frontend Design Flow

Design entry begins by partitioning the high-level specifications into separate modules, each containing a single CSP process. 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, 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 simplifies this partitioning and permits the designer to easily repartition without redesigning the communication scheme. Similarly, the CSP communication model ensures correct synchronization between the modules regardless of the partitioning chosen.

Hardware / software partitioning is performed by the designer, although partitioning tools could be added in the future. By simulating the entire design early in the design cycle, after partitioning, 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.

### C. Backend 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 FPGA

configuration files, called bitstreams, required for RTR. The backend implementation flow, shown in Figure 3, 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. 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 an 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 the hardware / software partitioning in the frontend design flow. Finally, different implementations of the application, from an algorithmic level, may be attempted and the entire flow repeated.

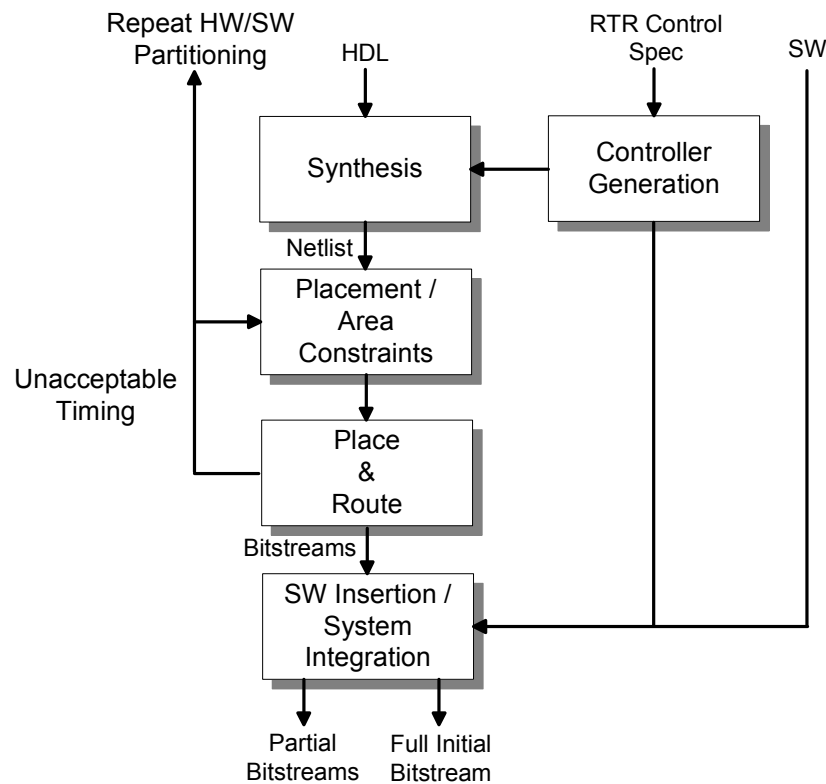


Figure 3. Backend Implementation Flow

For an RTR application, there are aspects to the design that cannot be specified easily using HDLs or vendor-specific constraint files. The HDL-focused methodologies, with their ASIC roots, treat the hardware as static and include no provisions for describing dynamic modules or connections. Previous projects have created their own methods for capturing the RTR-specific requirements [11]. In general these formats have not been published or, where they are available, are not suited to other development environments. This project addresses the deficiencies in existing work by developing a flexible RTR Control Specification Format (RCSF).

The intention is to not tie the designer to a specific development environment or device architecture and to facilitate the transfer of designs between architectures.

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:

- a list of dynamic modules, including connections,
- a list of embedded processors, including connections to other modules,
- a list of external resources required (memories, I/O, etc.), and
- configuration control information.

Much focus is placed on 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 link to software that handles reconfiguration or as involved as specifying the FSM for the controller.

The hardware implemented by this design methodology is shown in Figure 5 for a prototype design.

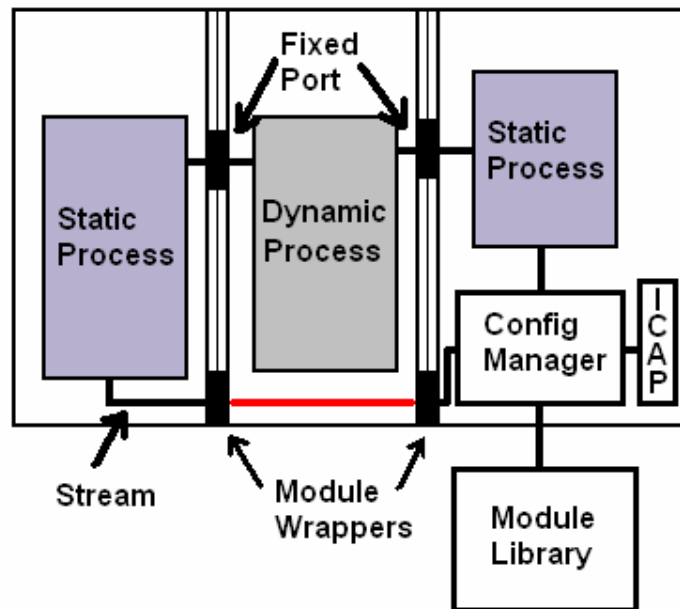


Figure 5. Generated Hardware Implementation.

### III. Impulse C Modifications

Impulse C is an ANSI C-based design language supporting HLS from C to RTL HDL. 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.

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. Preliminary modifications to the Impulse C simulation libraries have been performed that permit dynamic hardware to be simulated at a high level using the Impulse C design tools.

A methodology for configuration control has been selected that is centered on the idea of mutually exclusive processes. The designer identifies a set of processes 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 4. The figure describes an image processing application. The user may want to apply different filters to the image. 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. The modifications to Impulse C permit the description of these mutually exclusive processes and handle reconfiguration.

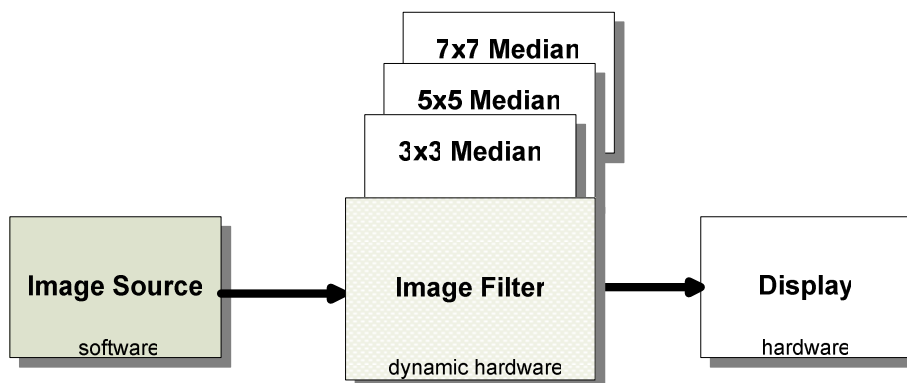


Figure 4. Mutually Exclusive Set of Processes.

#### IV. Conclusions

To facilitate the creation and exchange of RTR applications a high-level development methodology has been defined. This methodology consists of models of computation, communication, and configuration tailored for streaming applications. A frontend design flow captures the design using a HLS tool to create HDL. This architecture-independent frontend is coupled to a backend implementation flow using a special file format to capture the configuration management logic.

Modifications to Impulse C permit dynamic hardware designs to be described and simulated at a high level. In its current state the implementation flow involves manual steps to implement the Impulse C generated HDL in hardware. As these remaining stages are automated an image processing and an encryption application will be implemented with the high-level development environment and benchmarked against an existing dynamic hardware implementation flow.

#### References

- [1] J. Villasenor and W. Mangione-Smith, "Configurable computing," *Scientific American*, pp. 66–71, June, 1997 June, 1997.
- [2] R. Goering, "High-level synthesis rollouts enable ESL." *EETimes*, May 31st, 2004.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] S. Gehring and S. H.-M.-Ludwig, "Fast integrated tools for circuit design with FPGAs," In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 133--139, Feb 1998.
- [7] M. Eisenring and M. Platzner, "A framework for run-time reconfigurable systems," *J. Supercomputing*, vol. 21, no. 2, pp. 145–159, 2002.
- [8] C. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

- [9] J. A. Williams, N. W. Bergmann, and X. Xie, "FIFO communication models in Operating Systems for reconfigurable computing," in *Field-Programmable Custom Computing Machines*, 2005. pp. 277–278.
- [10] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Upper Saddle River, N.J.: Prentice Hall, 2005.
- [11] 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.