Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 901328, 10 pages
doi:10.1155/2008/901328

Research Article

Dynamic Hardware Development

Stephen Craven[1] and Peter Athanas[2]

[1] *Department of Electrical Engineering, The University of Tennessee at Chattanooga, Chattanooga, TN 37403, USA*
[2] *Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic and State University, Blacksburg, VA 24061, USA*

Correspondence should be addressed to Stephen Craven, stephen-craven@utc.edu

Received 31 March 2008; Accepted 12 August 2008

Recommended by Michael Hubner

Applications that leverage the dynamic partial reconfigurability of modern FPGAs are few, owing in large part to the lack of suitable tools and techniques to create them. While the trend in digital design is towards higher levels of design abstractions, forgoing hardware description languages in some cases for high-level languages, the development of a reconfigurable design requires developers to work at a low level and contend with many poorly documented architecture-specific aspects. This paper discusses the creation of a high-level development environment for reconfigurable designs that leverage an existing high-level synthesis tool to enable the design, simulation, and implementation of dynamically reconfigurable hardware solely from a specification written in C. Unlike previous attempts, this approach encompasses the entirety of design and implementation, enables self-re-configuration through an embedded controller, and inherently handles partial reconfiguration. Benchmarking numbers are provided, which validate the productivity enhancements this approach provides.

1. Introduction

Field-programmable gate arrays (FPGAs) are a class of integrated circuits that can be reprogrammed numerous times after manufacture to implement arbitrary digital circuits. While FPGAs always lag custom application-specific ICs (ASICs) in performance, the significantly reduced non-re-occurring engineering costs make FPGAs attractive for a variety of applications. However, with very few exceptions, an FPGA in a deployed design implements a single static design, behaving exactly as if it were a fixed-function ASIC.

The ability to reconfigure itself in a deployed product offers FPGAs a distinct advantage over ASICs. Whereas an ASIC must allocate area to implement every digital circuit the application requires, regardless of how infrequently it is actually exercised, an FPGA only need be sized large enough to support the circuits being active at any one time. The research community has demonstrated the benefits of swapping circuits in and out of an FPGA in such diverse applications as image detection [1], gene sequencing [2], video processing [3], network applications [4], and instruction set extension [5].

Vendor and tool support for the dynamic partial reconfiguration (PR) of an FPGA has suffered from severe limitations in the past. PR design flows were poorly supported and frequently broken. Device configuration architectures required that an entire configuration column be loaded just to change a single bit in the FPGA configuration. Self-re-configuration, through an internal configuration access port (ICAP), was limited to high-end devices, raising the cost of PR designs.

Recently, the PR landscape has experienced a change, driven in part by the growing importance of software defined radio (SDR), with its dynamic creation of radio waveforms. As the throughput requirements of SDR are impossible to meet with a processor and the configurability to implement arbitrary waveforms is beyond the capabilities of ASICs, the PR abilities of FPGAs are finally gaining tool support [6, 7]. The newer device families feature a configuration architecture that is more granular, increasing the speed and flexibility of PR [8]. Furthermore, PR capabilities have been extended to low-cost device families [9].

In spite of these trends, much work remains before PR design becomes an accepted practice. To develop a PR application using current tools, a designer must learn the intricacies of the target architecture and nuances of unfamiliar design flows. Lacking models and tools to abstract away the low-level specifics of each different architecture, every porting of a PR application to a different device requires that the design process start anew. Simulation of a PR design before implementation must be forgone, owing to a lack of simulator support, complicating verification and debugging.

Concurrent with the changes in the PR landscape has been a push towards electronic system-level (ESL) design. ESL design involves raising the level of abstraction that a designer sees from the register transfer level (RTL) to something higher than what traditional hardware description languages (HDLs) provide [10]. The research community has experimented with high-level languages (HLLs) to lift the abstraction level [11], and their results are paying off with a variety of commercial ESL tools now available [12]. Design specifications can now be captured in a multitude of formats from graphical [13] to C [14], and automatically converted to synthesizable HDL by commercial high-level synthesis (HLS) tools.

Recognizing the potential for HLS to drastically reduce the complexity of PR design, several researchers have described development environments utilizing some form of high-level design capture specifically tailored to PR design [15–17]. Notable limitations in these projects, though, hinder their ability to take advantage of recent trends in configurable computing. The reliance of many of these projects on an external host hinders development of embedded applications and ignores embedded processor capabilities of modern FPGAs. The use of outdated design entry techniques, such as JBits [18], shackles several projects to older architectures.

This paper describes a new approach to PR application development that leverages a commercial HLS tool, integrates embedded processors, and provides models of communication and reconfiguration. Previous publications have described the methodology [19] and the language extensions to an HLS toolset [20]. This paper focuses on the implementation and testing of the development flow, providing design and productivity results that validate this approach.

Section 2 provides an overview of previous attempts to raise the level of abstraction in PR design. An overview of the approach of this paper is presented in Section 3, with Section 4 detailing the implementation of applications and providing benchmarking results. Finally, conclusions are discussed in Section 5.

2. Background

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

Janus [16] was an early effort at a unified PR 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 in the same environment from JHDL, a Java-based structural hardware description language. Janus was developed under the coprocessor paradigm where the FPGA is essentially a slave to an external host processor. Partial reconfiguration and dynamic scheduling are not supported.

The PaDReH framework [21] 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. Little is provided to the designer in terms of tools or abstractions.

Synthesis and partitioning for adaptive reconfigurable computing systems (SPARCSs) [22] start 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 [23]. ISIS focuses on constraint-driven development and verification from a model-based approach. Tools automatically apply user-specified constraints to prune the design space. The development environment targets board-level designs comprised of heterogeneous computing elements (FPGAs, DSPs, processors, etc.), limiting the utility for FPGA-centric applications.

Recent work from Imperial College London defines abstractions of low-level details with an HLL-based approach to PR application development [15]. A modified form of C (RT-C) captures the design behavior at a high level, including configuration control. The RT-C is then translated into Handel-C [24], 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 [25]. The framework assumes a tight integration of the FPGA with a host processor running a controlling Java programme. This framework does little to facilitate the capture of configuration management or the incorporation of embedded processors.

The Caronte PR framework defines a high-level development environment targeting coprocessor applications [26]. Simulation of PR is possible via SystemC, with design entry via HDLs or Impulse C [27]. Caronte's use of Impulse C differs from the work presented in this paper in that Caronte merely uses Impulse C to produce HDL and not to capture the totality of the application including the configuration control. The bus-based communication of Caronte limits its applicability to streaming applications.

In addition to the projects described above, several researchers have explored the problem without producing a prototype design environment. Eisenring and Platzner's PR framework [28] describes a tool-independent design and implementation methodology in generic terms. Berkley's Stream Computations Organized for Reconfigurable Execution (SCORE) project [29] proposes a new FPGA-like architecture leveraging hardware pages to permit location-independent reconfiguration. While promising, no hardware has been produced.

These previous projects, summarized in Table 1, are each limited in important ways. Most assume a model of external configuration control, mandating the use of a host processor. For embedded application, this requirement is generally prohibitive. Many do not enable the use of partial reconfiguration. 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.

3. Approach

The goal of this project is to significantly reduce the effort required to deploy PR designs. To this end, a high-level development flow has been implemented that permits PR designs to be specified in C. Models of communication, computation, and reconfiguration have been defined that simplify design of streaming applications.

The development flow consists of a frontend, architecture-agnostic design flow, and a backend architecture-specific implementation flow. The design flow leverages an existing commercial HLS tool, modified to enable the capture and simulation of PR designs. By utilizing a commercial ESL tool, this work avoids the pitfalls of previous projects that relied heavily on outdated and unsupported tools such as JBits. The implementation flow is completely automated, encompassing floorplanning of the PR regions, insertion of a configuration controller, creation of the partial configuration bitstreams, and packaging of the configuration bitstreams for deployment.

Figure 1 presents the complete development flow, highlighting the exchange between the frontend and backend flows. To facilitate porting of designs to different architectures, the output of the frontend flow is completely architecture-agnostic. Due to variations in the configuration and clocking structure of different FPGA families, the backend flow may vary across architectures.

As conventional HDLs are not capable of capturing all aspects of PR designs, a reconfigurable computing specification format (RCSF) has been defined. The RCSF, expressed in XML, contains a list of reconfigurable modules, information concerning design connectivity, and the links to the HDL or SW that implements each module. A sample RCSF file is presented in Section 4.1. By editing this file, the designer can easily link to existing IP. A common use would be to replace a software test bench with the HDL that implements the actual

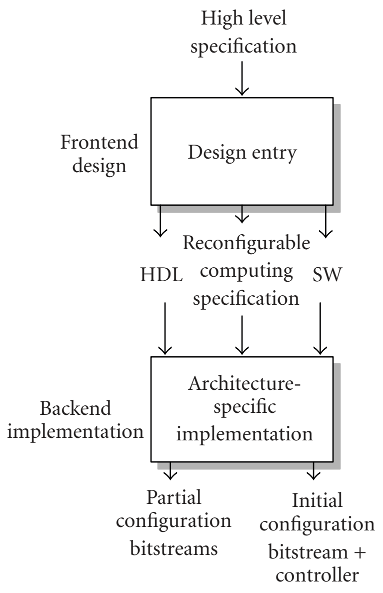

High level specification
Frontend design
Design entry
HDL
Reconfigurable computing specification
SW
Backend implementation
Architecture-specific implementation
Partial configuration bitstreams
Initial configuration bitstream + controller

Figure 1: Combined design and implementation flow.

interface to the application. Under this use model, a C model of the hardware IP could be leveraged to permit high-level simulation of the entire design early in the design cycle. This model would have to match the behavior of the hardware IP, but not the timing, as the high-level simulation is not cycle-accurate.

3.1. Abstractions

The models of computation and communication were selected to favor the traditional strengths of FPGAs, namely, streaming applications. Consisting of a repeatable schedule of computations operating on a steady flow of data, streaming applications are typically found in networking, signal processing, and cryptographic domains, all being strong suits of configurable logic. Multiple computational and communication models can accurately describe streaming applications, including several dataflow models and the communicating sequential processes (CSP) model [30]. In selecting an appropriate model, it was imperative that the actual functionality of hardware be captured and that commercial development tools support the model.

In CSP, an application is decomposed into a set of independently running processes, communicating only through unidirectional channels. Synchronization occurs during communication, with both the sender and receiver blocking until the transaction has completed. In contrast to some other dataflow paradigms, such as Kahn process networks [31] where communication occurs via infinitely deep FIFOs, CSP is directly implementable in hardware or software. Furthermore, tools and development environments exist supporting CSP design and implementation [14, 32, 33].

The implementation of an application using the CSP model of computation is straightforward. Communication channels can be created out of asynchronous FIFO buffers

Table 1: Previous PR development environments.

Project	Design entry	Model of computation	Architecture	Limitations
Janus	JHDL	Unspecified	Host + FPGA	No partial reconfiguration Requires host
SPARCS	Behavioral HDL	Dataflow	Host + FPGA	Requires macro library No partial reconfiguration
PaDReH	Multiple	Undefined	Standalone	Few defined tools
Model-Integrated	Dataflow graph	Dataflow	Independent	No partial reconfiguration Requires model library
RCAF	JHDL	Unspecified	Host + FPGA	No partial reconfiguration Requires host Few abstractions
Imperial College	RT-C	Dataflow	Limited by JBits	Requires host Manual translation
Caronte	Various	Coprocessor	Embedded proc	Limited automation

with minimal communication overhead. The FIFO-based communication permits easy integration with embedded processors as many Xilinx embedded processors feature fast simplex link (FSL) interfaces that are nothing more than asynchronous FIFO buffers linking the processor to peripherals [34].

To describe reconfiguration within a CSP model, the designer identifies a set of processes that are mutually exclusive in that only one of the set members is active in hardware at any one time. Figure 2 describes a cryptographic application where multiple decryption algorithms may be required, but never at the same time. Any process within the set of decryption cores may be selected for implementation, at which time the configuration manager reconfigures the FPGA to swap in the selected process. During reconfiguration, modules reading from or writing to the set undergoing reconfiguration will block until configuration is complete. This abstraction is similar to the swappable logic unit of Brebner [35] and the dynamic hardware modeling scheme of Luk [36].

This reconfiguration model enables the designer to utilize PR to extend an application breadth, by adding new functionality at runtime, or to extend an application 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.

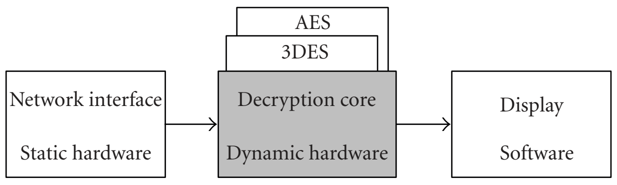


Figure 2: Set of mutually exclusive processes.

3.2. Frontend Design and Simulation

The language chosen for design entry is Impulse C, a commercial product of Impulse Accelerated Technologies, Inc. Impulse C [14] is an ANSI C-based language utilizing the same stream and process abstractions as Los Alamos National Lab's Streams-C work [11]. Based on the CSP model, Impulse C permits the application developer to describe hardware using a large subset of standard C. The CoDeveloper toolset performs high-level synthesis, translating Impulse C to synthesizable HDL.

Through an agreement with Impulse Accelerated Technologies, Inc., the CoDeveloper Impulse C application development environment has been obtained, along with the source code to the Impulse C simulation library. Modifications to the simulation library and corresponding extensions to the Impulse C language have been made permitting dynamic hardware to be simulated at a high level [20]. This modified language is referred to as DR Impulse C, highlighting its dynamic reconfiguration (DR) ability.

To describe PR applications in DR Impulse C, the programmer defines sets of mutually exclusive Impulse C processes. New Impulse C functions are utilized to create a set of reconfigurable processes and select a new dynamic process to execute in hardware. Applications described in DR Impulse C can be simulated by compiling the code in any C development environment. Each CSP process is spun off as a separate software thread communicating over shared buffers. PR is simulated by cleanly killing the executing thread and spinning off the new thread.

The frontend flow, shown in detail in Figure 3, consists of the CoDeveloper toolset for generating HDL from an Impulse C description, a preprocessor script for creating the RSCF file, and the GCC compiler for creating a simulation executable. Processes described in Impulse C can be marked

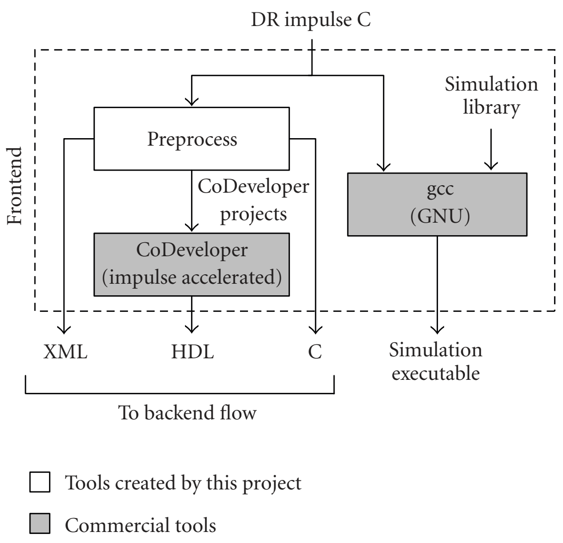


Figure 3: Frontend tool flow.

for hardware implementation, in which the CoDeveloper tools convert the corresponding code to an HDL, or can be targeted to an embedded processor. The implementation flow handles the mapping of software processes to specific processors available on the target platform.

3.3. Backend Implementation

The architecture-specific implementation flow accepts the RCSF file, HDL modules, and C code from the frontend. In addition, a board support package (BSP) must be specified, supplying all the platform-specific information required to produce a deployable design. The implementation tool flow, shown in Figure 4, integrates tools automating placement, HDL generation, and clock creation.

The postprocess tool parses the RCSF and BSP, generating a top-level Verilog wrapper that instantiates each module in the design, along with the PR control modules, MicroBlaze controller, and clocking structure. The Floorplanner utility is responsible for creating area constraints for each reconfigurable region of the FPGA. This tool accepts as input a list of the resource requirements of each set and a list of keep-out regions. The keep-out regions correspond to areas of the FPGA that must be available for peripherals or soft processors, such as regions near critical I/Os. In keeping with other FPGA floorplanning projects [37–39], Floorplanner uses a simulated annealing algorithm to find a near optimal minimum of a cost function.

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

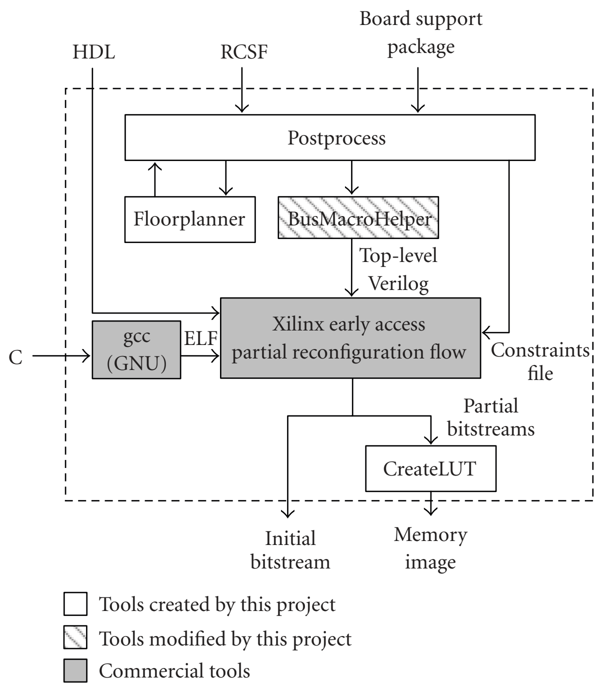


Figure 4: Backend tool flow.

tall, Floorplanner places all modules on configuration frame edges.

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

$$\begin{aligned} cost = {} & 10{,}000 * overlap + 10 * aspectError \\ & + waste + distance. \end{aligned} \tag{1}$$

Module overlap, contained in *overlap* as the sum of all overlapping CLBs, is weighted orders of magnitude higher in the cost function to ensure that no two PR regions will overlap. *aspectError* penalizes the placements for having a poor aspect ratio with the ideal aspect ratio being dependent on the architecture. Higher ideal aspect ratios are used for the Virtex-II families to minimize reconfiguration overhead. *waste* is a measure of extra resources within the placement that will not be utilized on the device. The *distance* variable represents the total distance between reconfigurable regions, and it is used to minimize routing delays between reconfigurable regions.

Producing partial configuration bitstreams currently requires an Xilinx-supplied patch to the standard Xilinx ISE toolset. Among other changes, this patch constrains the router to keep routes inside a reconfigurable region. These modified tools make up the Xilinx early access PR (EAPR) flow. The EAPR flow requires that special connection points, called bus macros, surround reconfigurable modules, providing a stable connection point to the static hardware.

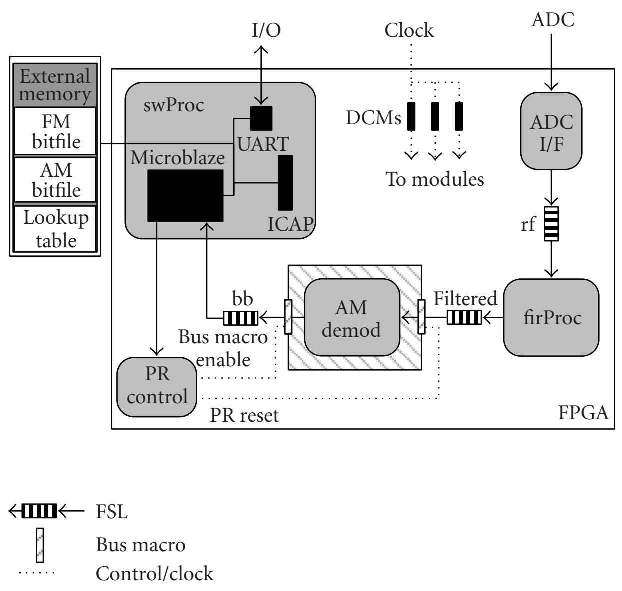


Figure 5: Final design implementation.

BusMacroHelper is a tool created for a related project that automatically inserts and places bus macros.

The CreateLUT tool creates a binary look-up table (LUT) that lists the size and location in memory of each partial bitstream enabling the configuration controller to find the desired partial bitstream. Additionally, the script concatenates the LUT and the partial bitstreams together into a single memory image to facilitate the automated download of the application to an FPGA.

Figure 5 presents an example implementation of a simple SDR application that may switch demodulation schemes. Several important aspects of this project are evident in the figure. The PR module *AM Demod* has been area-constrained to a specific location of the FPGA by the Floorplanner tool. All non-re-configurable modules are unconstrained, permitting the Xilinx tools to choose their optimum locations. All nonclock signals crossing the boundary between the static and PR regions must pass through a bus macro. As reconfiguration leaves the logic internal to a PR region in an undefined state, to stop the internal logic from producing random outputs that affect the rest of the system, the bus macro on the output of a PR region can be disabled. The tool flow automatically creates a PR control module for each PR region that disables the bus macros before reconfiguration and places any newly reconfigured module into a known good state by toggling the module reset line. Control of partial reconfiguration is handled by a MicroBlaze-based system running the user control code.

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

4. Results

A video processing application, representative of streaming applications that benefit from PR, is described in this section followed by a comparison of the results obtained with this development flow and the results obtained manually following the Xilinx EAPR flow [40].

4.1. Application development

A video processing demonstration has been implemented using this development flow in which a video stream is filtered in real time with one of several filters. A separate filter acts on each of the three colors (red, green, and blue) and each can be independently reconfigured to implement an edge detector, a median image filter, or a pass-through. The edge detector and median image filter operate on a 5×5 window of pixels. The application forgoes a full frame buffer, using a separate columns process to buffer five lines of pixels, presenting a column of five pixels to the filters.

The filters and control logic are all described in DR Impulse C. For high-level simulation, separate test processes are defined that load an input image from a Windows Bitmap (BMP) file and translate filters' outputs into a BMP, as shown in Figure 6. The filtered output images in Figure 6 were produced by this Impulse C simulation.

Before implementation, the application RCSF is edited to replace these Impulse C test benches with the interface logic for the video card and video DAC, which are a part of the BSP of the Xilinx Virtex-II Pro XUP development board. This edit involves the modification of only eight lines of XML code. The original RCSF file is shown in Figure 7. Each CSP process is linked to an implementation folder containing the HDL description. Connectivity is expressed by associating each port to a stream.

The implemented design (the layout of which is seen in Figure 8) encompasses 63% of an Xilinx xc2vp30. The filters operate at 57 MHz, sufficiently fast to support the incoming 640×480 video stream at 60 Hz. If implemented as a static design, the hardware would have to include nine separate filters, that is, three filters for each of the three colors. The total area required by all nine filters would be 1707 slices. Partial reconfiguration reduces the area requirements to three instances of the largest filter, consuming 1328 slices across three reconfigurable regions, thus resulting in an area saving of 379 slices due to using PR. Any additional filters added to the system would increase this area saving.

4.2. Benchmarks

To quantify the advantages and disadvantages of the high-level development environment, a set of applications was implemented in this environment and compared to implementations made following the Xilinx EAPR flow. To more accurately simulate real-world design practices, the Xilinx EAPR flow was scripted following the PR documentation [40]. All designs were created by an experienced hardware designer familiar with the Xilinx configuration architecture and EAPR flow. Note that the results presented below do

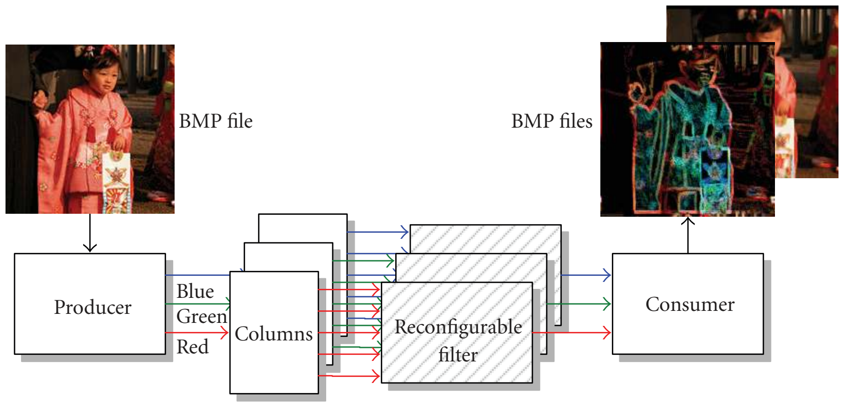
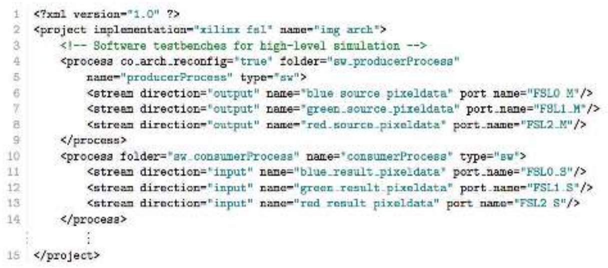


Figure 6: Video processing application.



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

Figure 7: RCSF file for simulated video processing design.

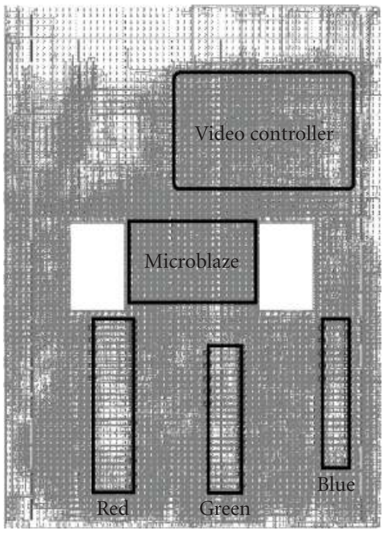


Figure 8: Floorplan of video processing implementation in an Xilinx xc2vp30.

not take into account the reduced skill set required by the high-level development environment. While some level of hardware experience is still required to create an application in DR Impulse C, it is significantly less than the low-level architecture-specific knowledge needed to follow the Xilinx EAPR flow.

The first application involved a reconfigurable coprocessor for an embedded MicroBlaze processor. This coprocessor, attached via an FSL interface, can be reconfigured to implement either a 32-bit integer divider or an integer square-root function. The descriptions for both functions were obtained from existing IP using the Xilinx Coregen tool and the OpenCores internet IP repository, in the case of the EAPR flow, and using example code provided with the Impulse C tools, in the case of this project's development environment.

The development time for both environments, from initial design description to working hardware implementation, was recorded. The PR region of the Xilinx EAPR flow was hand-placed, and it is 36% smaller than the Impulse C-based approach, owing to inefficiencies in HLS and automated floorplanning. Table 2 presents area and performance results at the module level. The Impulse C-generated divider compares well with the OpenCores divider, while the Coregen square-root function is significantly smaller than the Impulse C-generated module. The Impulse C-generated square-root function has a latency that is data-dependent. It should be noted that this high-level development environment can use existing IP and is not limited to Impulse C-created hardware though currently the implementation flow only supports IP with an FSL interface.

As presented in Table 3 for the integrated coprocessor application, the high-level development environment incurred a 71% penalty in average throughput and an 8% overall area penalty when compared to a manual implementation in the Xilinx EAPR flow. This throughput metric averages the best- and worst-case throughputs for the divider and square-root modules. The manual EAPR implementation ran the coprocessor at the system 100 MHz clock rate. The high-level development environment ran the coprocessor at 80% of the synthesis tool estimated clock rate for the slowest coprocess module. The performance penalty could be reduced by leveraging existing IP instead of using Impulse C-generated HDL. Additional gains are possible by dynamically modifying the clock rate of the coprocessor instead of running all coprocessors at the speed

Table 2: Coprocessor module performance benchmarks in an Xilinx xc2vp30.

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

Table 3: Coprocessor application performance benchmarks.

Environment	Area (slices)	Average Throughput (ops/sec)
High level (Impulse C)	3118	1.6 (10^6)
Xilinx EAPR	2883	5.6 (10^6)

Table 4: Coprocessor application productivity benchmarks.

Environment	Frontend (h)	Backend (h)	Total (h)
High level (Impulse C)	0.8	5	5.8
Xilinx EAPR	6.2	7.4	13.6

Table 5: Cryptographic module performance benchmarks in an Xilinx xc2vp30.

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

Table 6: Cryptographic application performance benchmarks.

Environment	Area (slices)	Throughput (blocks/sec)
High level (Impulse C)	2016	0.30 (10^6)
Xilinx EAPR	1632	0.58 (10^6)

Table 7: Cryptographic application productivity benchmarks.

Environment	Frontend		Backend	Total	
	(h)	(w/o MD5)	(h)	(h)	(w/o MD5)
High level	8.1	1	3.3	11.3	4.3
Xilinx EAPR	6.3	2.2	6.3	12.5	8.5

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

The high-level development approach netted a 57% reduction in overall development time, seen in Table 4. The frontend number indicates the time required to create the design description, whether in DR Impulse C or Verilog. The backend number represents the time required to take the design description through implementation, and includes any hardware debugging. While the DR Impulse C design bested the Verilog design for each metric, the majority of the productivity improvement came from the frontend design. Even with the EAPR flow leveraging existing IP, the time required to integrate this IP into a design was significantly greater than the time required to describe the application in Impulse C.

Cryptographic hash functions were used as a second benchmarking application. A reconfigurable region on the FPGA could be configured for either the MD5 or the SHA-1 standard. The hash functions were created from scratch using both Impulse C and Verilog. Area and performance numbers for each function are shown in Table 5. The Verilog-described SHA-1 consumed 12% more slices than the Impulse C design owing to the use of five independent memories to permit simultaneous access to the message data. This approach increases throughput at the expense of area. Had area been of primary concern, a Verilog design would have been smaller than the Impulse C-created hardware. The Impulse C MD5 and SHA-1 cores underperformed the Verilog cores by 39% and 63%, respectively.

Table 6 presents the performance results with the cryptographic modules integrating into the reconfiguration application. The high-level development environment imparts a 24% area penalty and a 48% performance penalty, compared to the conventional Verilog design.

The productivity advantage of the high-level development environment was hampered in this application by a bug in the Impulse C-generated hardware, as seen in Table 7. The time spent resolving this issue resulted in a 28% greater frontend design time for the high-level development environment than that for a Verilog-created design. If the MD5 design time was removed from consideration, the frontend design times for the high-level and conventional approaches are 1 and 2.2 hours, respectively. This 120% frontend design time improvement is more in line with the coprocessor productivity results. If the MD5 design and debug time are considered, the total development improvement of the high-level approach is 10%, while if the MD5 design time is excluded from both designs, the high-level productivity improvement increases to 49%, approximating the results for the coprocessor application.

While the performance and area results obtained from the HLS tool may limit its applicability to high-performance applications, this does not negate the utility of the presented dynamic hardware development environment. For designs with timing or area constraints that cannot easily be met with current HLS tools, the user is free to leverage HDL from other sources. This project's design and implementation flows offer many benefits even in the case of hand-coded HDL. The design flow permits high-level simulation of the entire design from a simple C model of each module. The implementation flow automates the creation of placement and area constraints, a configuration controller, and partial bitstreams.

It should be noted that the performance and productivity results would likely improve under a model-based high-level design environment. While Impulse C is currently used for design capture, other development tools that support a dataflow model may be leveraged with only slight modifications to the simulation mechanism of the tools. One advantage of Impulse C is its ability to synthesize random control logic. However, for straight signal processing applications, graphical high-level design tools, such as the Xilinx system generator, may be more appropriate. The defined interface between this project's design and implementation flows facilitates the use of multiple design entry methods.

5. Conclusion

The introduction of HLS techniques into the design of partially reconfigurable hardware for FPGAs can significantly reduce development time. The observed reductions in development time of approximately 50% would likely be greater for larger designs and for designers not being intimately familiar with an FPGA low-level configuration architecture. The resulting performance penalty may be acceptable for a variety of applications given the development time improvements and the significantly reduced skill set required to implement reconfigurable applications. By leveraging high-level development techniques, the full potential of FPGAs can be made easily available to the designer.

References

[1] K.-N. Chia, H. J. Kim, S. Lansing, W. H. Mangione-Smith, and J. Villasenor, "High-performance automatic target recognition through data-specific VLSI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 3, pp. 364–371, 1998.

[2] E. Lemoine and D. Merceron, "Run time reconfiguration of FPGA for scanning genomic databases," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95)*, pp. 90–98, Napa Valley, Calif, USA, April 1995.

[3] D. Ross, O. Vellacott, and M. Turner, "An FPGA-based hardware accelerator for image processing," in *Proceedings of the International Workshop on Field Programmable Logic and Applications on More FPGAs (FPL '94)*, pp. 299–306, Oxford, UK, September 1994.

[4] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet processing on the field programmable port extender (FPX)," in *Proceedings of the 9th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '01)*, pp. 87–93, Monterrey, Calif, USA, February 2001.

[5] M. J. Wirthlin and B. L. Hutchings, "Sequencing run-time reconfigured hardware with software," in *Proceedings of the 4th ACM International Symposium on Field Programmable Gate Arrays (FPGA '96)*, pp. 122–128, Monterey, Calif, USA, February 1996.

[6] J. Seely, "FPGA use in software-defined radios," EETimes, August 2004.

[7] "Xilinx, ISR offering SDR kit," EETimes, February 2006.

[8] Xilinx, Inc., "Virtex-4 configuration guide," 2007.

[9] Xilinx, Inc., "Spartan 3 generation configuration user guide," 2007.

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

[11] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, pp. 49–56, Napa Valley, Calif, USA, April 2000.

[12] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel, and A. George, "Survey of C-based application mapping tools for reconfigurable computing," in *Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD '04)*, Washington, DC, USA, September 2005.

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

[14] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*, Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[15] T. K. Lee, A. Derbyshire, W. Luk, and P. Y. K. Cheung, "High-level language extensions for run-time reconfigurable systems," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '03)*, pp. 144–151, Tokyo, Japan, December 2003.

[16] D. I. Lehn, R. D. Hudson, and P. M. Athanas, "Framework for architecture-independent run-time reconfigurable applications," in *Reconfigurable Technology: FPGAs for Computing and Applications II*, vol. 4212, pp. 162–172, Boston, Mass, USA, November 2000.

[17] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler, "Bridging the gap between compilation and synthesis in the DEFACTO system," in *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC '01)*, vol. 2624, pp. 52–70, Cumberland Falls, KY, USA, August 2001.

[18] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java based interface for reconfigurable computing," in *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Logic Devices Conference (MAPLD '99)*, pp. 1–9, Laurel, Md, USA, September 1999.

[19] S. Craven and P. Athanas, "A high-level development framework for run-time reconfigurable applications," in *Proceedings of the 9th Annual Conference on Military and Aerospace Programmable Logic Devices (MAPLD '06)*, Washington, DC, USA, September 2006.

[20] S. Craven and P. Athanas, "High-level specification of runtime reconfigurable designs," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*, pp. 280–283, Las Vegas, Nev, USA, June 2007.

[21] E. Carvalho, N. Calazans, E. Brião, and F. Moraes, "PaDReH—a framework for the design and implementation of dynamically and partially reconfigurable systems," in *Proceedings of the 17th Symposium on Integrated Cicuits and Systems Design (SBCCI '04)*, pp. 10–15, Pernambuco, Brazil, September 2004.

[22] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An integrated partitioning and synthesis system

for dynamically reconfigurable multi-FPGA architectures," in *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*, pp. 31–36, Orlando, Fla, USA, March-April 1998.

[23] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad, "Model-integrated tools for the design of dynamically reconfigurable systems," Tech. Rep., Institute for Software Integrated Systems, Vanderbilt University, Nashville, Tenn, USA, 2000.

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

[25] A. L. Slade, B. E. Nelson, and B. L. Hutchings, "Reconfigurable computing application frameworks," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 251–260, Napa, Calif, USA, April 2003.

[26] F. Ferrandi, M. D. Santambrogio, and D. Sciuto, "A design methodology for dynamic reconfiguration: the Caronte architecture," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 163, Denver, Colo, USA, April 2005.

[27] A. Antola, M. D. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, "A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse C and codeveloper," in *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL '07)*, pp. 221–224, Mar del Plata, Argentina, February 2007.

[28] M. Eisenring and M. Platzner, "A framework for run-time reconfigurable systems," *The Journal of Supercomputing*, vol. 21, no. 2, pp. 145–159, 2002.

[29] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE)," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL '00)*, pp. 605–614, Villach, Austria, August 2000.

[30] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[31] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

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

[33] A. Saifhashemi and P. A. Beerel, "High level modeling of channel-based asynchronous circuits using verilog," in *Proceedings of the Communicating Process Architectures Conference (CPA '05)*, vol. 63, pp. 275–288, Eindhoven, Netherlands, September 2005.

[34] J. A. Williams, N. W. Bergmann, and X. Xie, "FIFO communication models in operating systems for reconfigurable computing," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 277–278, Napa, Calif, USA, April 2005.

[35] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 77–86, Napa Valley, Calif, USA, April 1997.

[36] W. Luk, N. Shirazi, and P. Y. K. Cheung, "Modelling and optimising run-time reconfigurable systems," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '96)*, pp. 167–176, Napa Valley, Calif, USA, April 1996.

[37] L. Cheng and M. D. F. Wong, "Floorplan design for multimillion gate FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2795–2805, 2006.

[38] Y. Feng and D. P. Mehta, "Heterogeneous floorplanning for FPGAs," in *Proceedings of the 19th International Conference on VLSI Design Held Jointly with 5th International Conference on Embedded Systems Design (VLSID '06)*, pp. 257–262, Hyderabad, India, January 2006.

[39] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 605–612, Madrid, Spain, August 2006.

[40] Xilinx, Inc., "Early access partial reconfiguration user guide," March 2006.