VAPRES: A Customizable and Flexible Base Architecture for Partially Reconfigurable Systems

Ann Gordon-Ross and Abelardo Jara-Berrocal
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611
{ann, berrocal}@chrec.org

Abstract - *Partial reconfiguration (PR) enhances traditional FPGA-based high-performance reconfigurable computing by providing additional benefits such as reduced area and memory requirements, increased performance, and increased functionality as compared to non-PR systems. However, since leveraging these additional benefits requires specific designer expertise and increased design time, PR has not yet gained widespread usage. To alleviate some of PR's design challenges, we present VAPRES, a PR base architecture that provides a customizable and flexible platform for PR system and application design. VAPRES's key architectural contributions include customizable PR regions (PRRs) (e.g. size, shape, number of regions), a customizable inter-PRR communication architecture (e.g. number of and width of channels) offering reconfigurable and on-demand communication channel establishment, and seamless hardware module replacement. Finally, we present two design flows, system design and application design, to provide VAPRES design assistance.*

Keywords: partial reconfiguration, field programmable gate arrays (FPGAs), high-performance reconfigurable computing, embedded computing.

1. Introduction and Motivation

Field programmable gate arrays (FPGAs) have typically been used to implement custom data paths in reconfigurable computing, revealing benefits such as reduced power and area requirements as well as significant performance increases (commonly achieved speedups range from 10x-1000x when compared to equivalent software implementations [1][8][19]). Dynamically reconfigurable systems fully leverage the reconfigurability offered by FPGAs by exploiting the fact that not all hardware functionality is required at the same time, allowing mutually exclusive hardware modules to time-multiplex the FPGA fabric resources during runtime.

In order to realize these benefits, system designers are faced with a challenging task. Designing applications for FPGAs requires specialized design methodologies, hardware description language (HDL) knowledge, and use of specialized tool suites. In order to extract maximum performance speedups, designers must not only thoroughly understand the application's parallelism, but also be able to effectively modularize and synchronize application execution and data processing by partitioning the application into several/many communicating hardware modules. Runtime reconfiguration is the process of swapping out the hardware modules that have just finished executing with new hardware modules ready to execute. Unfortunately, runtime reconfiguration can impose lengthy performance overheads, as the entire system execution must halt as new hardware modules are loaded, even for small fabric changes. Full reconfiguration times (reconfiguring the entire FPGA fabric) can take on the order of hundreds of milliseconds [14][31] and may not be feasible for time critical applications. Finally, applications with many configurations can require excessive memory resources. A full bitstream (information needed to configure the entire FPGA fabric) is needed for each fabric configuration, even if two configurations have only small differences (redundant information must be stored in each full bitstream to encapsulate entire fabric operation). Careful consideration of all FPGA design challenges is critical to achieve maximum benefits as poorly designed systems can have detrimental side effects on power and performance.

Partial reconfiguration (PR) [28] enhances FPGA usability by partitioning the FPGA fabric into two main regions: the static region and the reconfigurable region. The static region contains all hardware functionality that will remain fixed during entire system execution (is never reconfigured). Reconfiguration is isolated to the reconfigurable region, which is further partitioned into several disjoint partially reconfigurable regions (PRRs). Each PRR can be individually reconfigured while all other PRRs and the static region remain operational. PRRs implement an application's hardware modules and runtime reconfiguration consists of loading/unloading hardware modules into PRRs. Hardware module switching is an enabling technology in novel operating system frameworks [9], fault tolerance [11], and artificial intelligence systems [5].

PR enhances FPGA benefits in several ways. PR enables reconfiguration without the need to halt entire system execution. This functionality can increase hardware time-multiplexing opportunities, leading to further reduced area and power requirements. Since PRRs can be individually reconfigured, hardware modules can even be prefetched into PRRs, significantly reducing or even eliminating runtime reconfiguration overhead by overlapping module execution with reconfiguration time. PR also reduces the system's memory requirements and reconfiguration time, as *partial*

bitstreams (information needed to reconfigure a PRR) are much smaller than full bitstreams.

PR's hardware module switching capabilities are particularly advantageous for reconfigurable stream processing systems (RSPSs). RSPSs are composed of a set of hardware and software modules (software modules execute on an embedded microprocessor core) connected together to transform a data input stream into a processed data output stream. The desired data stream transformations may be dependent stream characteristics, application requirements, or available resources. Since transformation goals may change mid-stream, RSPSs require mechanisms to dynamically switch stream-processing modules (i.e. apply a different filtering technique to a security monitoring video if a critical target is identified). However, for hardware module switching to be most advantageous, the switching process must be quick, incurring minimum stream processing interruption and no data loss.

Unfortunately, PR system design adds additional challenges and complexities and significantly increases system design time as compared to traditional non-PR FPGA system design. Current PR FPGA design tools (i.e. Xilinx Early Access PR flow [28]) are exceedingly complex, requiring system and application designers to have advanced knowledge of their target FPGA architecture. In addition, designer's must manually perform several time-consuming steps such as partitioning an application into the static region and one or more PRRs and creating the system floorplan by explicitly defining PRR physical locations and dimensions (e.g. size, height, and width). Application partitioning must also consider module size and granularity, module overlay (how and which modules will share PRRs), and module placement and scheduling. Special architectural considerations must also be addressed in order to provide hardware modules with the necessary inter-module and module-to-static region communication. The communication network must either be designed statically based on fixed module-to-PRR mappings or must provide an on-demand channel establishment architecture or network-on-chip (NoC) [3]. Finally, RSPSs require architectural support and a methodology to seamlessly swap hardware modules without interrupting data processing. Adding the complexity of PR design to the challenges already imposed with traditional FPGA design makes PR design even more susceptible to detrimental side effects on power and performance.

In this paper, we present architectures and methodologies aimed at increasing PR's amenability to a wider range of system designers. VAPRES (**V**irtual **A**rchitecture for **P**artially **R**econfigurable **E**mbedded **S**ystems) is a fundamental PR base architecture, which provides system designers with a highly customizable and flexible PR base system. Numerous architectural parameters provide fine-grained application specialization to meet varying application requirements. VAPRES includes a highly customizable and flexible inter-module communication architecture using switch-based on-demand communication channel establishment between arbitrary modules. In addition, VAPRES provides architectural support for local clock domains (each PRR can operate at a different clock frequency) and seamless module switching for RSPS applications. Finally, we present two PR design flows to leverage VAPRES: a PR system design flow and a PR application design flow.

2. Related Work

Conger et al. [7] formulated two methodologies to efficiently design and implement PR systems: a special-purpose and a multipurpose system design flow. The special-purpose system design flow targeted highly optimized PR systems and required complete application specification and behavior knowledge during system design time. In contrast, the multipurpose system design flow targeted the design of PR base systems for implementing a wide range of applications.

In the area of multipurpose PR design, Walder et al. [25] proposed a PR architecture for the Virtex-2. Floorplanning divided the FPGA fabric into multiple columns (vertical slots) to accommodate the hardware modules. A horizontal bus-structure crossing all columns using custom-designed bus macros provided inter-module and module-to-external processor communication. However, the authors did not devise a complete PR and soft-core processor integration implementation methodology.

Ullmann et al. [24] extended previous PR architectures to the Virtex-2 Pro by implementing a system-on-chip (SoC) consisting of a Microblaze processor, an internal configuration access port (ICAP) controller [28], and four user-definable PRRs. Williams et al. [26] and Bergmann et al. [4] proposed Egret, a similar architecture with an embedded Linux system running on the Microblaze processor. Both of these architectures did not support direct inter-module communication, forcing all inter-module communication to be routed through the Microblaze. The embedded processor core speed and amount of embedded RAM resources impacted both performance and amenability of applications that required inter-module communication.

To address the bottleneck imposed by routing all inter-module communication through a microprocessor, Bobda et al. [6] presented the Erlangen Slot Machine (ESM) for the Virtex-2. The ESM used a circuit-switched NoC (RMBoC) to enable each module to connect to any other module or system peripheral using a linear array of switches. This one-dimensional architectural layout provided unrestricted module-to-PRR mapping. Architectural parameters included number and width of communication links, but switches were restricted to only one input and one output port for module connections. Switches used a centralized FIFO and arbiter to receive all connection requests and dynamically established dedicated communication paths between modules. However, the architecture did not leverage flow control, which is problematic when a modules exhaust buffer memory resources. Finally, the modules and communication architecture were required to operate at the same clock frequency. RMBoC achieved an operating frequency of 99 MHz and required 3407 slices (10% of device usage) on a Virtex-2 6000 for a design with four switches and four communication links between switches.

Sedcole et al. [22] developed Sonic-on-a-Chip, an image processing PR architecture for the Virtex-2 Pro and Virtex-4. This work introduced the idea of separating the PR base system

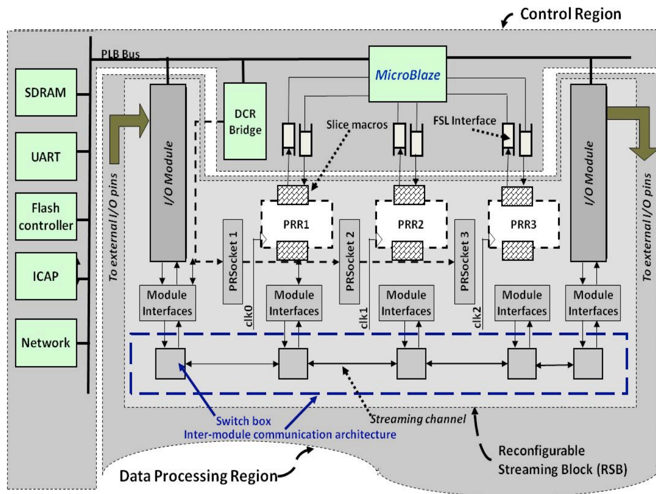

Figure 1: VAPRES architectural layout showing a single reconfigurable streaming block (RSB)

design from application design. The base system design created the base PR architecture, partitioned into the static region and PRRs, whereas application design consisted of partitioning any application to run on a pre-defined PR base system. This separation introduced a base system reuse mechanism, enabling application designers to develop multiple applications using a single base architecture. The authors defined several desirable key aspects for PR base system design including modularity, high levels of abstraction, and orthogonalisation concerns, such as the separation of inter-module communication and module computation. Sonic-on-a-Chip's communication architecture allowed dynamic streaming channel establishment directly between PRRs by allocating slots on a time-multiplexed bus. However, due to long routing delays, the reported bus clock frequency was 50 MHz.

Koch et al. [20] presented ReCoBus, a PR architecture for the Virtex-2 Pro (based on a reconfigurable multibus [10] adapted for FPGA implementation [6]), and an associated builder tool to automate system design. A Microblaze processor connected several fine-grained PRRs (each PRR was only one CLB wide) through a horizontal bus named ReCoBus that included interfaces for standard bus protocols such as AMBA (Advanced Microcontroller Bus Architecture) from ARM [13], Wishbone from Opencores [33], and CoreConnect from IBM [15]. Architectural parameters included PRR height, number of PRRs, and data bus width. Since ReCoBus used custom-slice macros to accommodate horizontal bit and address lines crossing all PRRs, bus propagation delay degraded the maximum attainable clock frequency. The maximum clock frequency for a system with 60 PRRs was 50 MHz.

Sudarsanam et al. [23] developed PolySAF, a PR architecture for reconfigurable processing based on systolic kernels for the Virtex-4 using a Microblaze connected to a set of PRRs. PR enabled dynamic replacement of systolic kernels inside PRRs. The communication architecture used a multiplexed FIFO-based interface for communication between adjacent PRRs (PRRs placed next to each other in the floorplan) and between the Microblaze and the PRRs. An analytical performance model evaluated the effect of the number of PRRs and PRR sizes (which determines the complexity of the systolic kernels that can be loaded inside the PRRs) on an application's performance.

3. VAPRES Architecture

VAPRES [17] is a multipurpose PR FPGA SoC composed of a soft-core Microblaze processor in the static region connected to a set of PRRs. VAPRES introduces several novel architectural features, such as the ability to operate hardware modules at independent and configurable clock frequencies (local clock domains). Figure 1 depicts the VAPRES architectural layout comprised of two fundamental regions: the *controlling region* and the *data processing region*.

3.1. Controlling Region

VAPRES's controlling region contains a Microblaze processor and a set of static peripherals. The controlling region is responsible for three main functions: controlling data processing region operation via PRSockets (Section 3.3.2), performing system-level functions such as reading hardware module bitstreams from external memory and performing PR via the ICAP, and executing software modules.

3.2. Data Processing Region

The data processing region contains one or more reconfigurable streaming blocks (RSBs). Each RSB has several PRRs (each RSB can have a different number of PRRs), I/O modules (IOMs), and an inter-module communication architecture called SCORES – a **S**calable **CO**mmunication architecture for **RE**configurable **S**ystem [18]. IOMs and SCORES are located inside the static region of the system.

Figure 1 depicts a sample VAPRES system with one RSB containing three PRRs and two IOMs. PRRs and IOMs within each RSB communicate using SCORES. IOMs directly interface to external I/O pins or peripherals (i.e. ADCs, DACs, sensors, etc.). PRRs interface with the Microblaze processor through asynchronous FSL (fast simplex link) interfaces. For each switch box-PRR or switch box-IOM pair, a PRSocket allows the Microblaze processor to control switch box, hardware module, IOM, and module interface operation. PRSockets contain one *device control register* (DCR) [27] and additional interfacing logic. The DCR connects as a slave peripheral to the Microblaze processor through a PLB-to-DCR (Processor Local Bus) bridge.

3.2.1. SCORES Architecture

Figure 2 depicts the top-level design of the SCORES communication architecture. SCORES is composed of a linear array of switches (one switch is highlighted in gray shading). Each switch has a unique X coordinate indicating its horizontal position inside the linear array. Switches communicate with neighboring switches (*Kl* and *Kr*) and computing module interfaces (*Ki* and *Ko*) through bidirectional communication links between their input and output ports.

In VAPRES, a computing module can either be a PRR or an IOM. Computing modules attach to switches through two types of module interfaces. *Consumer Interfaces* connect a computing module's input port to a switch's local output port

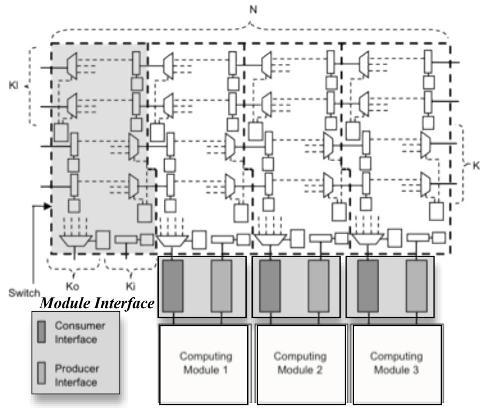

Figure 2: Top SCORES communication architecture. In VAPRES, a computing module can either be a PRR or an IOM

(*Ko*). *Producer Interfaces* connect a computing module's output port to a switch's local input port (*Ki*).

Dynamically established *Data Streaming Routes (DSRs*) enable data transmission between two computing modules. These dedicated routes provide high throughput and low latency data transmission. For each DSR, we refer to the *producer* as the module sending data and the *consumer* as the module receiving data. A computing module can be both a *producer* and *consumer* simultaneously.

SCORES is a highly parametric communication architecture with six tunable architectural parameters: *N*, *W*, *Kr*, *Kl*, *Ki*, and *Ko* (Figure 2). *N* represents the number of switches in the linear array. *W* is the width of the communication links and switch input and output ports. *Ki*, *Ko*, *Kr*, and *Kl* represent the number of local input ports, local output ports, right output and left input ports, and left output and right input ports, respectively, for each switch. Thus, a switch has *Kl* and *Kr* communication links to the left and right neighboring switches, respectively.

3.2.2. Switch Architecture

Figure 3 depicts the block level diagram of the SCORES switch architecture. The switch uses distributed arbitration and contains two main block types: input blocks and output blocks. Input and output blocks enable data to flow into and out of the switch, respectively. These blocks encapsulate and manage a switch's input and output ports. External connections between neighboring switch's input and output blocks, and internal connections between input and output blocks collectively enable inter-module communication.

Output blocks are units responsible for controlling switch output ports. Output blocks are classified into three different types: left, right, and local output blocks. Left and right output blocks are responsible for all left and right output port management for the switch, respectively. To enable horizontal data transmission through the linear switch array, a switch's left output blocks are connected to the neighboring switch's right input blocks and vice versa. To enable internal data transmission through a switch, left output blocks are internally connected to right input blocks and vice versa. Each switch has only one left and one right output block, but each block can be connected to multiple output ports. Local output blocks are responsible for local output port management, which connect the switch to computing module interfaces.

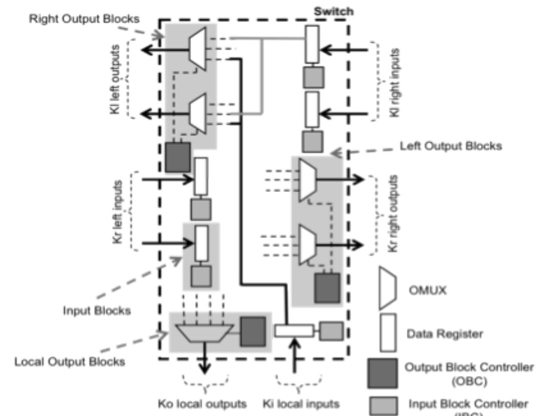

Figure 3: SCORES switch architectural components

Input blocks are similar to output blocks in that there are three types of input blocks: left, right, and local input blocks. Each block type serves a similar purpose as the associated output block type. In contrast to left and right output blocks, there is one input block for each input port. Depending on the input block type, input blocks are connected with a subset of output blocks. Left input blocks are internally connected to the right output block and to all the local output blocks. Right input blocks are internally connected to the left output block and to all the local output blocks. Finally, each local input block is connected to both the left and right output blocks.

3.3. Local clock domains (LCDs)

Local clock domains (LCDs) enable a PRR to regulate data processing throughput. For example, in a system with a series of digital filter hardware modules and a fixed processing throughput requirement, some hardware modules may require more processing cycles and a higher clock frequency than other hardware modules. To provide this configurability, the VAPRES static region and PRRs are independently clocked, and each constitutes a separate LCD. The Microblaze sets LCD clock frequencies using the PRSocket. The asynchronous FIFOs inside the FSLs and module interfaces provide isolation between the PRRs and the static region LCDs.

In order to implement PRRs as LCDs on the Virtex-4, PRRs must be constrained to fit inside a group of adjacent Virtex-4 local clock regions [12]. Virtex-4 local clock regions vertically span sixteen CLBs and horizontally span half of the FPGA fabric. To ensure successful system implementation, local clock regions used by different PRRs may not intersect. In addition, we used Virtex-4 regional clock buffers (BUFRs) [12] to implement buffered clock signals inside each PRR and Virtex-4 clock multiplexer primitives (BUFGMUX) to generate the clock signals feeding the BUFR's clock inputs. Since a Virtex-4 BUFR can only drive the two regional clock nets in the same local clock region where the BUFR is located and the two clock nets in the adjacent local clock regions (up to three local clock regions), the PRR height must be no greater than 3x16=48 CLBs. The Microblaze configures PRR clock frequencies during runtime using the PRSocket, which connects to the BUFGMUX select signals. We implemented the multiple clock signals feeding the BUFGMUX primitives using the Virtex-4

Figure 4: A SCORES Streaming Data Channel (SDC)

DCM (Digital Clock Manager) and PMCD (Phase Matched Clock Divider) primitives.

4. VAPRES Operation

VAPRES's operation consists of communication establishment to construct DSRs between communicating modules, computing module interface operation to transmit data across the DSR, RSPS assembly and module encapsulation, and seamless hardware module swapping. Whereas VAPRES operation also requires a runtime hardware manager to orchestrate these processes, the runtime hardware manager is the focus of our current work.

4.1. SCORES Communication Channel Establishment

Establishing a complete DSR from a producer module to a consumer module consists of establishing numerous *Streaming Data Channels* (*SDCs*) spanning the switch path between the two modules. SDCs, illustrated in Figure 4, connect a switch with neighboring switches and computing modules. The link consists of two opposite flowing data channels and three handshaking signals. The SDC is the main channel and transmits data from a switch or computing module output port to the connected input port. For a data channel of *W* bits, the two most significant bits (MSBs) of the SDC are reserved for signaling. The MSB is the *Write Enable* (*WR_EN*) and indicates that the producer is transmitting a word. The second MSB is the *End of Stream* (*EOS*) and indicates that the producer has completed data transmission and that the DSR can be released. The remaining W-2 least significant bits (LSBs) of the SDC carry data. The *Stream Feedback Channel* (*SFC*) is a single signal, *Remote FIFO Full*, which indicates that the consumer FIFO is full, and therefore the producer must pause data transmission. The handshaking signals (*REQ* and *ACK*) establish and release a DSR.

When a producer requests DSR establishment with a target consumer interface, the producer writes an *Address Header* to the SDC of its producer interface. The *Address Header* is composed of two fields, an *X coordinate* and a *local identifier*. The *X coordinate* indicates the horizontal location of the target switch connected to the consumer interface. The *local identifier* indicates the specific local output port to use between the target switch and the consumer interface. Use of a *local identifier* enables computing modules to separate different data stream types to different input ports. *X coordinate* and *local identifier* widths depends on the *N* and *Ko* architectural parameters

After the producer interface receives the *Address Header*, the producer interface writes the *Address Header* to the connected switch's input port and asserts *REQ*. The switch selects an arbitrary left or right (direction determined by the *X coordinate* field) output port that is not already assigned to a

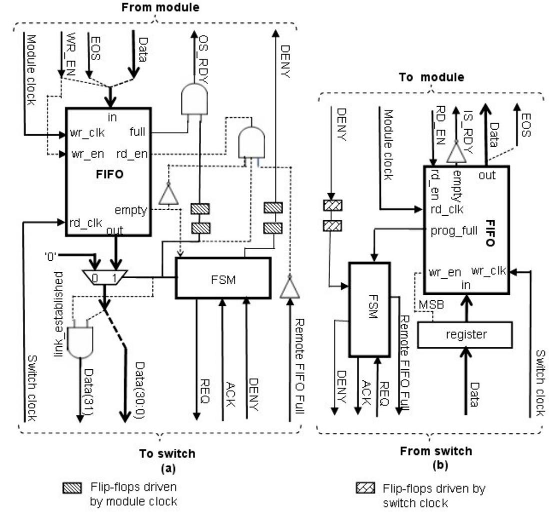

Figure 5: Module interfaces: (a) producer interface, (b) consumer interface.

DSR and forwards the *Address Header* and asserts *REQ* on this output port. The connection between the input and output port is now reserved for this DSR. This similar process repeats as the *Address Header* propagates through neighboring switches until the *Address Header* reaches the target switch, in which case the *Address Header* is forwarded to the target consumer interface.

When the target consumer interface receives a *REQ*, the consumer interface enters an *Established Connection State* and replies with a positive *ACK*. This *ACK* propagates through the switch array back to the producer interface, traversing the reserved input/output port connections at each switch. When the producer interface receives the asserted *ACK*, a DSR has been established between the producer and consumer interfaces.

After DSR establishment, data can be transmitted between the producer and the consumer as a continuous low latency pipelined stream because our switch design uses only one register at each input port instead of a large, high latency FIFO. The DSR remains established as long as the producer interface asserts *REQ*. A producer interface deasserts *REQ* when the producer interface detects assertion of the *EOS* flag from the producer module.

4.2. Computing Module Interface Operation

Computing module interfaces connect computing modules to a SCORES switch. These module interfaces are based on dual-clocked FIFOs. These FIFOs buffer data and enable clock domain isolation between the communication architecture and the computing modules. By separating clock domains, each computing module can run at an independent and optimized clock frequency.

We created FIFOs using the Xilinx Coregen FIFO Generator 4.3 [29]. This tool enables customization of both FIFO depth and width (of which the width was set to match the switch's *W* architectural parameter). Xilinx Coregen tool allows FIFOs to be implemented using distributed memory or embedded BlockRAMs.

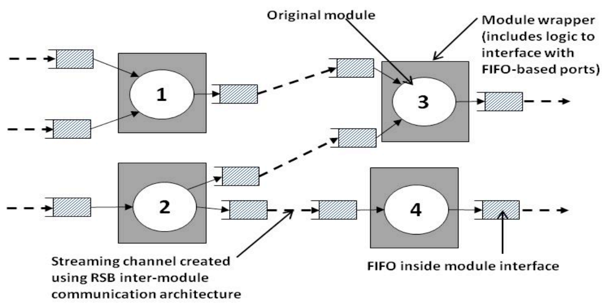

Figure 6: Kahn process network inside a VAPRES RSB

Figure 5 (a) illustrates the detailed producer module interface architecture. Signals between the module output port and the producer interface are: *Data_in*, *WR_EN* (Write enable), *OS_RDY* (Output Stream Ready), *EOS* (End of Stream), and *DENY*. The FIFO's data input (*in*) and output (*out*) are the combination of *data_in*, *EOS*, and *WR_EN*. The producer module asserts *WR_EN* to begin data transmission to the producer interface. The producer module asserts *EOS* when data transmission has completed, allowing the producer module interface to release the DSR.

Figure 5 (b) illustrates the detailed consumer module interface architecture. Signals between a module input port and the consumer interface are: *data_in*, *RD_EN* (Read Enable), *IS_RDY* (Input Stream Ready), and *EOS* (End of Stream). The module asserts *RD_EN* to enable reading from the FIFO. The module interface asserts *IS_RDY* when the FIFO contains data waiting to be read. Since the MSB of *data_in* indicates *WR_EN*, *data_in* coming from the switch's local output port is written into the FIFO if *WR_EN* is asserted.

4.3. RSPS Runtime Assembly and Module Encapsulation

The process of *RSPS runtime assembly* consists of placing hardware modules in PRRs and establishing on-demand inter-module communication through SCORES. RSPSs assembled using SCORES approximates a Kahn Process Network (KPN), a widely used model for implementing streaming digital signal processing systems [22]. Hardware modules map to KPN nodes and module interface FIFOs and FSLs map to KPN stream buffers. Figure 6 shows a possible mapping of nodes and buffers of an example KPN inside a VAPRES RSB.

Hardware modules read/write data from/to module interfaces and FSLs through FIFO-based ports, which offer advantages over alternative NoC architecture interfaces [2][5]. First, hardware modules can read/write to/from FIFOs using a simple, well-known communication protocol instead of the complex addressing and synchronization schemes common in NoCs. FIFOs transparently implement blocking-read and blocking-write synchronization mechanisms when hardware modules detect FIFO empty and FIFO full signals, respectively. Secondly, FIFO-based ports increase the system design abstraction level, enabling application designers to develop hardware modules independently of VAPRES architecture details. However, application designers must encapsulate hardware modules (the *original modules*) inside special *module wrappers* to connect the original module's input and output ports with the external FIFO-based ports.

4.4. Hardware Module Switching Methodology

Efficiently leveraging PR for hardware module switching presents several challenges. First, PR imposes stream processing interruption because the reconfigured PRR must halt operation as the new hardware module is loaded. However, since the new hardware module is downstream from other hardware modules, the upstream hardware modules must halt operation. Since PRR reconfiguration can take on the order of hundreds of milliseconds [9][16], this stream processing interruption may be unacceptable. In some cases, module interface FIFOs can buffer data to alleviate stream processing interruption. However, for RSPSs with high stream processing throughput requirements, FIFOs may fill quickly, resulting in significant stream processing delays. Second, in many RSPSs, a new hardware module's initial operational state must match the replaced hardware module's current operational state. Additionally, the replaced hardware module may have computed dynamic variables required by the new hardware module. The capability to save and restore state registers inside hardware modules enables the operational state and dynamic variables to be transferred from the replaced hardware module to the new hardware module.

VAPRES addresses these challenges using a custom hardware module switching methodology. Figure 7 exemplifies this methodology using a digital filter example where circled numbers indicate intermediate steps. The system is composed of one RSB with one IOM and two PRRs. P0, p1, and p2 denote the producer module interface FIFOs and c0, c1, and c2 denote the consumer module interface FIFOs. R0, r1, and r2 denote the FSL links flowing towards the Microblaze and t0, t1, and t2 denote the FSL links flowing towards the PRRs/IOMs.

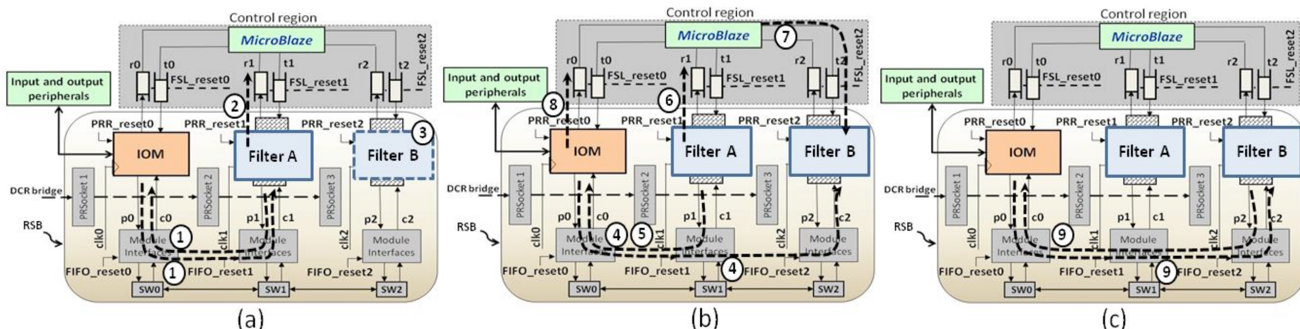

Figure 7: Switching digital filters (hardware modules) inside a VAPRES RSB: (a) Initial RSPS operation and placement of filter B in the second PRR; b) Intermediate RSPS operation and detection of the end of stream condition; (c) Final RSPS operation. Circled numbers indicate intermediate steps.

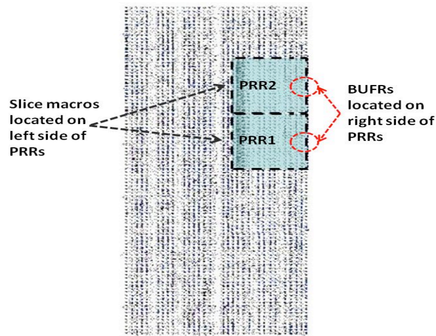

Figure 8: VAPRES prototype floorplan on the VLX25 indicating location of regional clock buffers (BUFRs) and slice macros.

This example assumes that prior to RSPS operation, the Microblaze placed filter A inside the first PRR and configured switch boxes SW1 and SW2 to establish DSR between p0 and c1 and between p1 and c0.

The RSPS initially operates as follows: filter A receives streamed input data from the IOM and sends the processed streamed output data back to the IOM (step 1). While filter A processes data, filter A periodically sends monitoring information about input data characteristics through r1 to the Microblaze processor (step 2). The Microblaze evaluates this monitoring information to determine if filter B would better meet the design constraints (i.e. reduced power, higher precision, etc.). If filter B is determined to be more appropriate, the Microblaze reconfigures the second PRR to store filter B while filter A continues data processing (step 3).

After the second PRR reconfigures to filter B, the Microblaze configures the switch boxes to release the DSR between p0 and c1, in addition to establishing a new DSR between p0 and c2 (step 4). Filter A continues processing the remaining data words present in the consumer interface FIFO. After processing the remaining data, filter A sends a special end of stream word to the IOM (step 5) and the state register values to the Microblaze through r1 (step 6). The Microblaze initializes filter B using the state register values (step 7). After the IOM detects the special end of stream word arriving from c0, the IOM informs the Microblaze that filter A operation has completed by writing a message through r0 (step 8). The Microblaze configures the switch boxes to connect p2 and c0, completing hardware module switching (step 9).

This hardware module switching methodology overlaps module operation with PRR reconfiguration, which avoids stream processing interruption. The new hardware module is placed outside the current RSPS processing path and begins operation only after PRR reconfiguration has finished.

5. Analysis and Results

To validate and evaluate the overheads and performance of our system, we evaluate both a complete VAPRES prototype with fixed a SCORES configuration and SCORES separately using numerous architectural parameters.

5.1. VAPRES Prototype

We implemented a VAPRES prototype system on a Xilinx ML401 evaluation board to test system functionality and evaluate the reconfiguration time for individual PRRs. Figure 8 depicts the FPGA fabric layout consisting of one RSB with two PRRs and one IOM (sufficient for functionality testing purposes). We customized the inter-module communication architecture with two 32-bit channels flowing both left and right between switch boxes and one 32-bit module input port and one 32-bit module output port connecting PRRs to switch boxes. Module interface FIFOs and FSL links were implemented using Virtex-4 BlockRAM, which buffer 512 32-bit words. The Microblaze processor and switch boxes executed at 100 MHz. In addition, PRRs were constrained to fit inside separate Virtex-4 local clock regions and contained 640 slices, which spanned sixteen vertical CLBs and ten horizontal CLBs. We point out that these PRR sizes are relatively small, and larger PRRs might be required for applications with larger hardware modules, but however are sufficient for testing purposes.

The VAPRES static region (including the Microblaze soft-core processor and SCORES) required 9,421 slices (approximately 86% of the VLX25), of which SCORES required only 1,020 slices (approximately 9% of the VLX25 device). We generated both static and partial bitstreams with the Xilinx Early Access Partial Reconfiguration Flow [28]. Hardware module partial bitstreams were stored as files in external flash memory.

We evaluated PRR reconfiguration time for two of the VAPRES functions using the Microblaze xps_timer peripheral. The first function transfers the partial bitstream from the external compact flash memory to the ICAP port and the second function transfers a partial bitstream from an array stored in SDRAM to the ICAP port. Reconfiguration of a single PRR using a bitstream from flash memory required 1,043,388,614 clock cycles (1.043s) of which transferring the partial bitstream from flash memory to the ICAP BRAM buffer accounted for 95.3% of the time and writing the partial bitstream to the ICAP accounted for 4.7% of the time. Reconfiguration of a single PRR was reduced to 71,944,572 clock cycles (71.94 ms) when using a bistream from SDRAM (the partial bitstream was copied from flash memory to an array in the SDRAM at system startup). Since partial bitstream size will directly influence reconfiguration time and thus system performance, a focus of our future work includes analyzing the tradeoffs between resource fragmentation and system performance for large verses small PRRs. In addition, we will explore mechanisms to prefetch bitstreams to the SDRAM to further reduce reconfiguration overhead for systems where bitstream storage requirements exceeds SDRAM resources.

5.2. SCORES Evaluation

We implemented our SCORES communication architecture, switch, and module interfaces as highly parametric VHDL soft cores, providing the architectural parameters: *N*, *W*, *Kr*, *Kl*, *Ki*, and *Ko*. FIFOs, implemented using one embedded BlockRAM, stored 512 32-bit words. The target device was a Virtex 4 XC4VLX25 [30] and system simulation was performed using Modelsim 6.2 SE [21].

Given the massive configurability of SCORES due to the numerous architectural parameters, we wrote a Perl script to

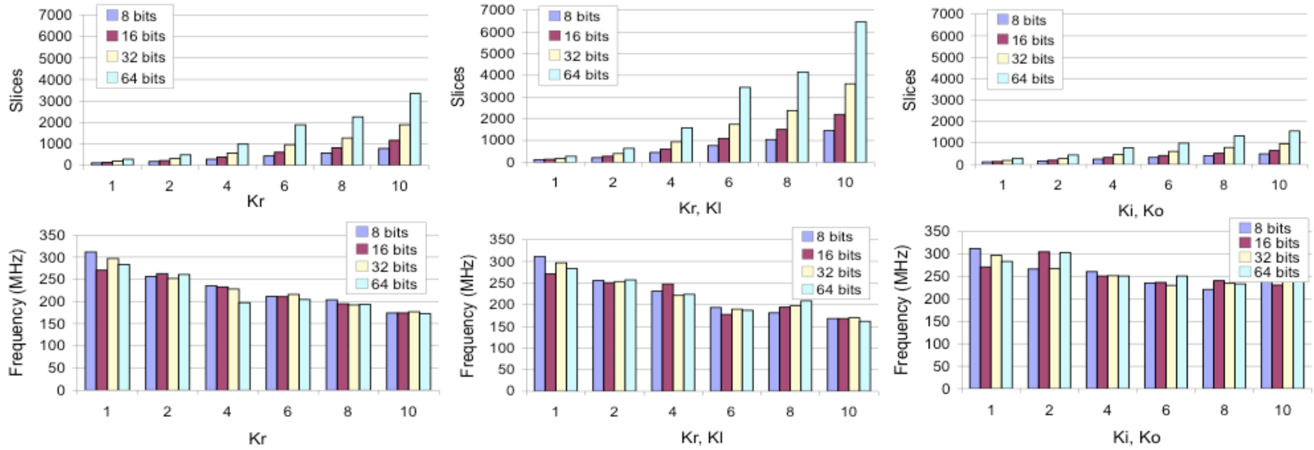

Figure 9: Area in slices (top row) and maximum clock frequency (bottom row) verses varying architectural parameters for channel widths W = 8, 16, 32, and 64 bits.

execute Xilinx synthesis and implementation tools (ISE 10.1) [32] for a standalone switch of varying configurations. These configurations enabled architectural parameter impact evaluation on selected performance metrics such as slice utilization and maximum clock frequency. In a real scenario, *Kr*, *Kl*, *Ki* and *Ko* would be specialized to the target application. We measured maximum clock frequency after place-and-route using the Xilinx Trace static timing analysis tool with no clock constraint (*trce –a –u*).

Figure 9 shows area usage in slices (top row) and maximum attainable clock frequency (bottom row) versus varying architectural parameters for channel widths W = 8, 16, 32, and 64 bits for a single switch. The first, second, and third columns vary *Kr, Kr* and *Kl,* and *Ki* and *Ko,* respectively. We consider the case in which only *Kr* is varied to account for applications in which most data flow occurs in only one direction such as DSP or image processing. Figure 9 (top row) shows low switch area overhead, which scales well due to a small cross point matrix. For example, the area overhead for a sample system configuration, W = 32, *Kr* = 2, *Kl* = 2, *Ki* = 1, and *Ko* = 1, is only 399 slices, which accounts for 1.62% of the XC4VLX25. Doubling the channel width from W = 32 to W = 64 (with the same system configuration *Kr* = 2, *Kl* = 2, *Ki* = 1, and *Ko* = 1) increases slice usage by only 60% to only 639 slices, revealing a sublinear increase in area verses channel width.

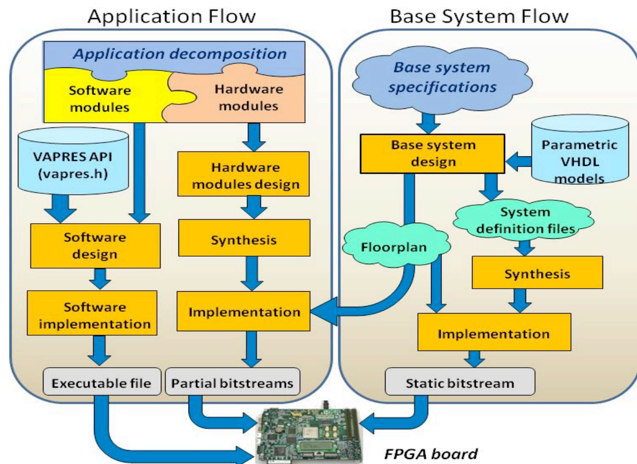

Figure 10: VAPRES design and implementation flows

Maximum clock frequency is a very important metric since it determines the maximum data throughput achievable by SCORES. Given a data word of length W, (with the two MSBs reserved for *WR_EN* and *EOS*) the peak data throughput in Gbps for SCORES is:

$$\text{data_throughput (Gbps)} = (W - 2) * \text{max_frequency}$$

Figure 9 (bottom row) shows that for all test configurations, the operating frequency ranges from 161 MHz to 311 MHz. Therefore, the data throughput peaks at 161*(32-2) = 4.8 Gbps for an SDC width of W = 32 bits, which is competitive with previous work.

6. VAPRES System Design and Implementation

Creating an FPGA-based PR SoC using the VAPRES architecture requires two design flows: (1) the base system flow assists system designers in creating a VAPRES base system (Figure 10 right), and (2) the application flow assists application designers in creating applications to run on the VAPRES base system (Figure 10 left).

6.1. Base System Flow

In the base system flow's first step, the system designer determines the *base system specifications* by specializing the VAPRES architectural parameters. In order to leverage reusability and architectural specialization, Figure 11 shows the VAPRES architectural parameters for a single RSB. Architectural parameters include the maximum number of PRRs (N), communication channel width (w bits), number of one-way communication channels between switch boxes (*kr* channels flowing to the right and *kl* channels flowing to the left), and the number of input channels (*ki*) and output channels (*ko*) between each PRR and the connected switch box. This architectural specialization supports a wide variety of hardware module and application requirements and enables system

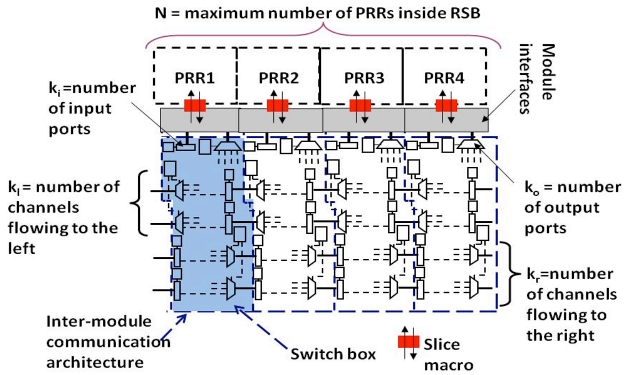

Figure 11: Sample RSB with the following architectural parameters: N=4, w=32, kr=2, kl=2, ki=1, ko=1

designers to balance resource utilization with communication flexibility.

In the *base system design* step, the system designer designs the base system floorplan and creates the *system definition files*. System definition files include the VHDL code modeling the static region, a Microprocessor Hardware Specification (MHS) file defining the system structure for the Xilinx EDK tool *platgen*, a Microprocessor Software Specification (MSS) file defining the base system build process for the Xilinx EDK tool *libgen*, and a User Constraints File (UCF) representing the system floorplan.

To ensure that the VAPRES floorplan is suitable for the Virtex-4, system designers must ensure that each PRR fits inside one to three adjacent local clock regions and that local clock regions used by different PRRs do not intersect. In general, three adjacent local clock regions are required for PRRs containing large hardware modules, but large PRRs can increase resource fragmentation (wasted resources when a hardware module requires fewer resources than a PRR provides). An alternative solution constrains PRRs to fit within one local clock region, and hardware modules that require more resources than a PRR provides can span multiple adjacent PRRs. Finally, the *synthesis* and *implementation* steps generate the base system's static bitstream.

6.2. Application Flow

After downloading the base system's bitstream to the FPGA device, an application designer designs applications for the base system. The application designer decomposes an application into software and hardware modules using hardware/software co-design techniques. After decomposition, the hardware and software modules follow two separate flows. During the *software module design* flow, the application designer writes the application software that will run on the Microblaze processor. In order to assist the application designer in writing software modules for the VAPRES systems, Application Program Interface (API) functions provide low-level system functionality (Table 1).

During the *hardware module design* flow, the application designer designs the hardware modules and hardware module wrappers. Application designers are insulated from low-level PR design tasks involving PRR definition, floorplanning, and other base system implementation details. However, the application designer must consider the number of, data-width, and type of input and output ports connected to each hardware module. A hardware module's input and output port type can be an FSL slave (reads data from an FSL link), an FSL master (writes data to an FSL link), a consumer port (reads data from a consumer interface), or a producer port (writes data to a producer interface). During the application flow, only logic associated with each hardware module is synthesized and placed and routed, as the base design logic remains unchanged. This isolation between the application flow and the base system flow reduces synthesis and place and route times, which otherwise can be exceedingly high during the iterative development and testing stages of large, complex designs.

7. Conclusions and Future Work

In this paper, we designed and prototyped VAPRES – a multipurpose PR FPGA SoC. VAPRES enables intense architectural specialization to meet design constraints through numerous architectural parameters and local clock domains. A novel hardware module switching methodology enables dynamic system reconfiguration without stream processing interruption. In order to assist system and application designers in developing VAPRES base systems and applications, we formulated two customized design flows. Future work includes additional design support in the form of scripting tools for system floorplan definition and system definition file creation.

8. Acknowledgements

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. We gratefully acknowledge tools provided by Xilinx.

9. References

[1] J. Bakos, P. Elenis, J. Tang. FPGA Acceleration of Phylogeny Reconstruction for Whole Genome Data. 7th IEEE International Symposium on Bioinformatics & Bioengineering, 2007

[2] E. Beigne, P. Vivet. Design of on-chip and off-chip

Function	Purpose
int vapres_CF2ICAP(XHwIcap *hwicap, Xuint8* filename) ;	Transfers a partial bitstream stored as a file in CF memory to ICAP port
int vapres_array2ICAP(XHwIcap *hwicap, char* bitstream) ;	Transfers partial bitstream stored as a ***bitstream*** array in SDRAM to ICAP port.
int vapres_CF2array(char* bitstream, int* size, Xuint8* filename) ;	Transfers a partial bitstream file from CF memory to a ***bitstream*** array in SDRAM. Array size is returned on argument ***size***.
int vapres_module_clock (int num, bool enable);	Enables the regional clock buffer (BUFR) for HW module identified by ***num***
int vapres_module_reset(int num, bool assert);	Resets the HW module identified with number ***num***
int vapres_module_write(int num, int value);	Writes ***value*** to hardware module input identified with number ***num***
int vapres_module_read(int num, int value);	Reads a ***value*** from the ***num***-th hardware module identified with number ***num***
int vapres_establish_channel(comm._state* current_state, Xuint8 prrx, Xuint8 prry)	Establishes a streaming channel between PRRs identified with number ***X*** and ***Y***

Table 1: Sample VAPRES API functions.

interfaces for a GALS NoC architecture. 12th IEEE International Symposium on Asynchronous Circuits and Systems, 2006. March 2006

[3] L. Benini and G. De Micheli. Networks on chips: A new SOC paradigm. In *IEEE Computer*, pages 70-78, 2002

[4] N. Bergmann, J. Williams, and P. Waldeck. Egret: A Flexible Platform for Real-Time Reconfigurable System-on-Chip. International Conference on Engineering of Reconfigurable Systems and Algorithms, 2003.

[5] C. Bobda. Introduction to Reconfigurable Computing. Architectures, Algorithms and Applications. Springer, 2007

[6] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. Increasing the Flexibility in FPGA-based Reconfig-urable Platforms: The Erlangen Slot Machine. IEEE Conference on Field-Programmable Technology (FPT), 2005

[7] C. Conger, A. Gordon-Ross. A. George. FPGA Design Framework for Partial Run-Time Reconfiguration. ERSA, 2008.

[8] T. Court, M. Herbordt. Families of FPGA-Based Accelerators for Approximate String Matching. ACM Microprocessors & Microsystems, v. 31, Issue 2, 2007

[9] E. El-Araby, I. Gonzalez, T. El-Ghazawi: Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing. ACM Trans. on Reconf. Technology and Systems (TRETS), 2009

[10] H. A. ElGindy, A. K. Somani, H. Schroeder, H. Schmeck, and A. Spray. RMB - A Reconfigurable Multiple Bus Network. Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), 1996.

[11] J. Emmert, C. Stroud, M. Abramovici. Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration. FCCM, 2000

[12] E. Eto. BUFR in partial reconfigurable modules. Xilinx WP 344, 2008.

[13] D. Flynn. AMBA: Enabling Reusable On-chip Designs. Micro, IEEE, vol. 17, no. 4, pp. 20 – 27, 1997.

[14] R. Garcia, A. Gordon-Ross, and A. George. Exploiting Partially Reconfigurable FPGAs for Situation-Based Reconfiguration in Wireless Sensor Networks. FCCM 2009.

[15] A. Goel and W. R. Lee. Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core. Design Automation Conference, 2000. Proceedings 2000. 37th, 2000, pp. 196 – 200.

[16] R. Hymel, A. D. George, H. Lam. Evaluating Partial Reconfiguration for Embedded FPGA Applications. HPEC, 2007

[17] A. Jara-Berrocal and A. Gordon-Ross. VAPRES: A Virtual Architecture for Partially Reconfigurable Embedded Systems. IEEE/ACM Design, Automation and Test in Europe (DATE), March 2010.

[18] A. Jara-Berrocal and A. Gordon-Ross. SCORES: A Scalable and Parametric Streams-Based Communication Architecture for Modular Reconfigurable Systems. IEEE/ACM Design, Automation and Test in Europe (DATE), April 2009

[19] V. Kindratenk, D. Pointer, A case study in porting a production scientific supercomputing application to a reconfigurable computer. FCCM 2006

[20] D. Koch, C. Beckhoff, and J. Teich. ReCoBus-Builder - A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAS. FPL 2008.

[21] Mentor Graphics, http://www.mentor.com, 2008

[22] P. Sedcole, P. Cheung, W. Luk: Run-Time Integration of Reconfigurable Video Processing Systems. IEEE Trans. VLSI Syst. 15(9), 2007

[23] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, A. Dasu Dynamically Reconfigurable Systolic Array Accelerators: A case Study with EKF and DWT algorithms. In-print IET Comput. and Digit. Tech., 2010

[24] M. Ullmann, B. Grimm, M. Hübner, J. Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. IEEE Parallel and Distributed Processing Symposium, 2004

[25] H. Walder, S. Nobs, M. Platzner. XF-board: A Prototyping Platform for Reconfigurable Hardware Operating Systems. ERSA 2004

[26] J. Williams and N. Bergmann. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip. Engineering of Reconfigurable Systems and Algorithms (ERSA) 2004.

[27] Xilinx Inc. Device Control Register Bus (DS402, v. 2.9), May 2005

[28] Xilinx Inc. Early Access PR User Guide (v1.1). March 2006

[29] Xilinx. FIFO Generator 4.3 Datasheet. DS317. March 24, 2008

[30] Xilinx Virtex 4 User Guide. UG070 v2.6. December 1, 2008

[31] Xilinx Inc. Virtex 4 Configuration Guide (UG071), January 2006

[32] Xilinx Inc., http://www.xilinx.com, 2008

[33] X. Xing, C. Zezong, J. Jing, and K.Hengyu. Porting from Wishbone Bus to Avalon Bus in SoC Design. in Electronic Measurement and Instruments. (ICEMI), 2007.