FACULDADE DE INFORMÁTICA
PUCRS - Brazil
http://www.inf.pucrs.br

Design Frameworks and Configuration Controllers for Dynamic and Partial Reconfiguration

*Ewerson Carvalho, Frederico Möller,
Fernando Moraes, Ney Calazans*

TECHNICAL REPORT SERIES

Number 042

June, 2004

Contact:

calazans@inf.pucrs.br

http://www.inf.pucrs.br/~calazans

E. Carvalho holds a M.Sc. degree obtained at the PPGCC/FACIN/PUCRS in 2004. He is Research Assistant at PUCRS/Brazil since April, 2004. His main research interests are digital systems design, fast prototyping, run-time reconfiguration, and reconfiguration control.

F. Möller is an undergraduate student on Computer Engineering at UFRGS. He is a research assistant at GAPH research group, funded by an undergraduate research grant from CNPq. His main research interests are digital systems design and fast prototyping.

F. Moraes works at the PUCRS/Brazil since 1996. He is a professor since August 2003. His main research topics are digital systems design and fast prototyping, digital systems physical synthesis CAD, telecommunication applications, hardware-software codesign. Dr. Moraes is a member of the Hardware Design Support Group (GAPH) at the PUCRS.

N. Calazans works at the PUCRS/Brazil since 1986. He is a professor since 1999. His main research topics are digital systems design, fast prototyping, and applications, reconfigurable systems, and embedded systems. Dr. Calazans is the head of the Hardware Design Support Group (GAPH) at PUCRS.

Design Frameworks and Configuration Controllers for Dynamic and Partial Reconfiguration

Ewerson Carvalho, Frederico Möller, Fernando Moraes, Ney Calazans

Pontifícia Universidade Católica do Rio Grande do Sul (FACIN - PUCRS)
Av. Ipiranga, 6681 - P 30/Bl 4 - 90619-900 - Porto Alegre - RS - BRASIL
{ecarvalho, frebm, moraes, calazans}@inf.pucrs.br

Abstract. Dynamically reconfigurable systems, especially those where the hardware can be changed during runtime, have the potential to provide hardware with flexibility similar to that of software. At the same time, they may lead to better performance and smaller system size. However, there is a clear lack of support devices, tools and design flows adequate for such systems. This paper presents the state of the art of design frameworks to dynamic and partial reconfiguration and proposes a framework named PaDReH. Also discuss the problems for enabling dynamically reconfigurable systems is the unavailability of efficient methods to control the hardware configuration process and, proposes a configuration controller, called RSCM. This controller is implemented and validated in Virtex-II Xilinx FPGAs.

1. Introduction

Along the previous decade, it is possible to notice a considerable increase of the interest on reconfigurable computing [1]. The potential flexibility provided by reconfigurable hardware is relevant and has the potential to increase the lifetime of products. Similar to software systems that constantly receive updates, hardware implemented with reconfigurable devices can put this strategy to good use to preserve product utility for longer time. In addition, the time available to execute design flow continually decreases because of market pressures. Therefore, the use of configurable technology coupled to the massive reuse of intellectual property can accelerate System-on-Chip (SoC) design time [2], decreasing the time-to-market of technological products.

An attractive feature of using reconfigurable computing is the possibility to implement a whole system in less silicon than its nominal minimal requirement, developing the concept of *virtual hardware* [3]. The use of *Dynamically Reconfigurable Systems* (DRS) design techniques has potential to save resources while reducing the system area overhead. This happens because they allow that parts of the systems not needed in some time interval be removed from the hardware to make room for another part of the system, required at that same interval. On the other hand, potential drawbacks of DRS are the performance penalty induced by the often long reconfiguration times and the area overhead to implement the hardware responsible for controlling the reconfiguration process.

Also, enabling DRSs requires strong support that is not yet available [1]. This support is composed by tools to enable the use of DRSs and infrastructure to implement them. Tab. 1 presents a summary of the main features requiring better support to enable that DRS develop its potential to become a mainstream technology.

Tab. 1 – Main features requiring better support to enable DRS design and implementation.

Support lacking	**Tools**	- DRS design - DRS verification
	Infrastructure	- DRS-enabling devices - Modules to control dynamic reconfiguration process - Standard interface for reconfigurable modules

The implementation of DRS usually assumes the availability of an infrastructure composed by specific modules for system control and operation. Among these modules, one is responsible to manage the system reconfiguration process, the *configuration controller*. The main objective of the present work is to propose a solution to this lack of infrastructure for DRS, evaluating the relative costs of the approach.

The rest of this paper is organized as follows. Section 2 contains a brief review of the state of the art on DRSs. Section 3 presents a survey of previous DRS frameworks propositions. Section 4 describes the main features of the PaDReH framework, here proposed. The state of the art on configuration controller models and implementations is presented in Section 5. From there, a model of configuration controller is proposed, in Section 6. Next, a case study used to validate the proposed model is presented in Section 7. A set of implementation results using the case study is analyzed in Section 8, while Section 9 presents some conclusions and directions for future work.

2. DYNAMIC AND PARTIAL RECONFIGURATION

Several approaches were proposed to enable the use of dynamic and partial reconfiguration as revised in [1][4][5]. This Section is intended as a specific discussion on three topics relevant to this work: (*i*) commercial devices enabling DRS design and implementation; (*ii*) tools to generate partial bitstreams; (*iii*) methods used to interconnect IP cores in DRSs.

2.1 Commercial Devices Enabling DRS

There are still very few commercially available semiconductor devices that enable the use of dynamic and partial reconfiguration. Atmel Inc. produces two series of partially reconfigurable devices: AT6000 and AT40k. Another vendor, Xilinx Inc. commercializes four FPGA device series supporting dynamic and partial reconfiguration: Virtex, VirtexE, Virtex-II and Virtex-II Pro series. Since Atmel devices offer a maximum gate count of around 50,000 equivalent gates, these are useless to implement complex DRS systems in a single device. On the other hand, Xilinx devices can reach up to 10 million equivalent gates, justifying its choice for use in this work.

2.2 Tools and Techniques for Partial Bitstreams Generation

The generation of partial reconfiguration files, also called partial bitstreams, is a crucial task for DRS development. The production of partial bitstreams with current FPGA tools is basically a manual, complex and error-prone process. However, some tools and techniques to automate the generation of partial bitstreams can already be found on the research literature as JbitsDiff [6], Jbits [7], PARBIT [8] and JPG [9]. As another example of techniques, Dyer et al. [10] present three alternatives for partial bitstream generation. The first employs a flow executed with vendor tools only, allowing *difference based* manipulations where just very small, localized portions are changed at a given moment. The second alternative is to use the JBits class library to generate partial *bitstreams*, promoting a higher degree of abstraction in the process. The last alternative combines the two previous approaches. This mixed flow uses vendor tools for synthesis task, and JBits to generate partial bitstreams. This last alternative is pointed by the authors as the one leading to the best results.

2.3 IP Core Interconnection Schemes in DRS

Palma [11] suggests a method to generate the interconnection among partial bitstreams representing arbitrarily complex IP cores using a bus-based structure. The method is partially automated, but is limited by the difficulties to fine control the routing in Xilinx FPGA designs. Another technique for interconnecting dynamically replaceable cores in FPGAs has been recently proposed by Xilinx [12]. This technique, based on the Xilinx Modular Design flow establishes a set of steps to generate partial bitstreams. This is followed by a set of steps to incorporate each of these partial bitstreams in an FPGA at runtime. Communication between fixed and reconfigurable parts or among reconfigurable parts is provided through the use of use pre-designed macro cells named *bus-macros*, furnished by Xilinx. The flow is still plagued by many manually executed steps for enabling correct generation of partial bitstreams and correct definition of communication paths in the border of modules.

3. FRAMEWORKS TO DRS DESIGN AND IMPLEMENTATION

Several approaches have been proposed to organize the design, implementation and maintenance of reconfigurable systems. This Section reviews several relevant propositions in this theme.

The Brass project proposes SCORE (Stream Computations for Reconfigurable Execution) [13], a computation model based on the organization of reconfigurable systems around the virtualization of three main hardware concepts: paged reconfigurable hardware, page communication through the use of streams, and storage.

The Janus framework [14] targets the development of reconfigurable systems composed by stages implemented either in software and hardware and based on JHDL, a set of classes built upon Java.

Edwards and Green [15] present an integrated run-time environment to support partially and dynamically reconfigurable systems based on the discontinued Xilinx XC6200 FPGA family. They propose FSS (FPGA Support System) a system to support loading and unloading of hardware tasks on FPGAs. The software applications are developed in C++ and hardware tasks are described in VHDL.

The Model-Integrated Development Environment for Adaptive Computing (MIDE) project [16] has as goal to develop high-level system design tools for implementing dynamically reconfigurable systems using adaptive computing technology. MIDE provides tools that allow designers to construct and exercise graphical models of the systems, used to create executable computational structures (software and hardware), and a runtime environment. The system is aimed at embedded systems of weapons like missile guiding systems and uses DSP processors coupled to Virtex Xilinx FPGAs.

The design environment CHAMPION [17] provides automatic retiming to match paths through the dataflow and performs multi-FPGA design partitioning, without knowledge of the specific architecture implementation.

Eisenring et al. [18] proposes a run-time reconfiguration of FPGA computing resources, where system behavior and architecture are represented as a *problem graph*, and an *architecture graph*, respectively. Tool support is provided to map the nodes in the problem graph to nodes in the architecture graph, assign problem graph nodes to FPGA configurations, and schedule the execution order of problem graph objects and configurations.

The Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems (SPARCS) [19] takes a design specification at the behavioral level in the form of a task graph, divides and schedules the tasks on the target architecture and maps the tasks to individual FPGAs in multi-FPGA systems.

4. PADREH FRAMEWORK FOR DRS

PaDReH is a framework to design and implement DRSs. This framework is intended to enable obtaining advantages from the use of dynamic and partial reconfigurable hardware technology. The general characteristics of this framework are as follows.

First, PaDReH is initially addressed to deal only with the DRS hardware design flow only. This is the same decision followed by SPARCS, Einsenring et al. approach, and CHAMPION, but distinct from the choice in Janus and MIDE. The reasoning behind the decision is that most of the complexity of DRSs lies on the hardware side, and it is necessary to extensively experiment with hardware concepts before delving into software implications of DRS.

Second, PaDReH is targeted to support partial and dynamic reconfiguration of single FPGAs. This is justified by the fact that current FPGAs are already capable of embedding entire complex systems, and that FPGA advances are fast enough for this to stay valid in the long term. This choice is the same for the systems MIDE and the one proposed by Edwards et al., and addressing it is planned by Einsenring et al. The opposite approach, suggested by CHAMPION, SPARCS and currently supported by Einsenring et al., DRSs implemented in multiple FPGAs, is useful for low-end applications. This occurs once multiple FPGAs typically present reduced performance and cost, while single FPGA DRSs are adequate for high-end applications.

Third, regarding the underlying implementation technologies, PaDReH advocates the use of higher level abstractions for design capture and validation through the use of languages like SystemC. Also, initial development of back-end DRS support employs Xilinx Virtex families FPGAs devices (Virtex, Virtex2 and Virtex2-Pro). The justification for these choices is the emerging of SystemC as a de facto standard for addressing hardware modeling at abstraction levels above RTL, and the fact that Virtex

FPGA families are the only commercial devices to support System-on-Chip integration capability. These choices can be compared to the use of Khoros by the CHAMPION system, JAVA by Janus, and C++/VHDL in the approach of Edwards et al. On the support device side, most reviewed systems claim to use or to be considering the use of Virtex families.

4.1 The PaDReH System Structure

The PaDReH system is composed by three module sets, as depicted in Fig. 1.

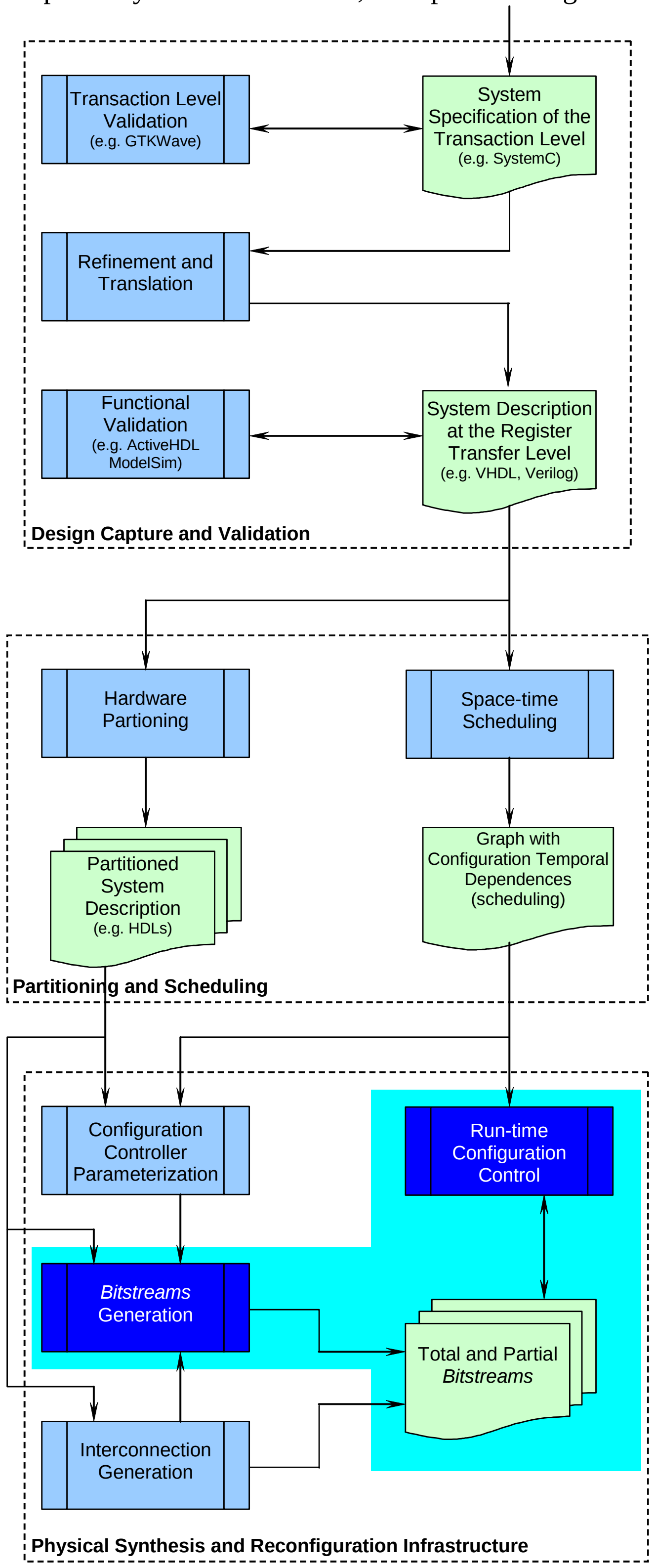

Transaction Level Validation (e.g. GTKWave)
System Specification of the Transaction Level (e.g. SystemC)
Refinement and Translation
Functional Validation (e.g. ActiveHDL ModelSim)
System Description at the Register Transfer Level (e.g. VHDL, Verilog)
Design Capture and Validation
Hardware Partioning
Space-time Scheduling
Partitioned System Description (e.g. HDLs)
Graph with Configuration Temporal Dependences (scheduling)
Partitioning and Scheduling
Configuration Controller Parameterization
Run-time Configuration Control
Bitstreams Generation
Total and Partial *Bitstreams*
Interconnection Generation
Physical Synthesis and Reconfiguration Infrastructure

Fig.1 - PaDReH framework design, verification and implementation flow for DRS.

The first module set is named *Design Capture and Functional Validation*. It is responsible for the description and validation of DRSs at high abstraction levels and translation from these to the *Register Transfer Level* (RTL) of abstraction.

Next, the *Partitioning and Scheduling* module set is responsible for the generation of files that describe the DRS behavior. These files are usually represented in a hardware description language (HDL). The same files are transmitted to the module set *Physical Synthesis and Reconfiguration Infrastructure*. This module set is responsible for the generation of configuration files implemented as total and partial bitstreams, along with the spatial and temporal system partitioning, as defined in the second module set. It is also responsible for inserting the parameterized configuration controller module in the system, according to the specific DRS characteristics. The generation of the physical interconnection implementation (e.g. bus or network-on-chip) among cores of the DRS is also performed in this module set.

The shaded area in Fig. 1 represents the parts of the framework that are currently supported through the use of automated methods and tools. Other parts of PaDReH have been specified and are currently conducted by combinations of manual application of the method and/or with the help of commercial tools.

5. Configuration Controllers

One of the main problems for enabling reconfigurable systems is the unavailability of efficient methods to control the hardware reconfiguration process. A configuration controller commands which reconfigurable IP core(s) must be inserted on the reconfigurable device at any moment, and which must be removed. It executes tasks similar to those of a loader of an operating system. This module is responsible for loading configurations to execute on the reconfigurable hardware, according to a defined task scheduling. This Section reviews models and implementations of configuration controllers.

Configuration controller models and implementations have different characteristics, but most share common parts. It is possible to identify the following parts in most configuration controllers: (*i*) Configuration store, (*ii*) Reconfiguration monitor, (*iii*) DRS status registers, (*iv*) Configuration loader and (*v*) Configuration relocation.

Tab.2 compares all configuration controller models and implementations that could be found in the available literature. The last column in the Table shows the characteristics of the configuration controller proposed here, detailed in next Section. The most sophisticated model, proposed by Lysaght et al. [20] allows the use of advanced scheduling strategies, preemption and allows the manipulation of compacted or encrypted data. The simplest model, proposed by Shirazi et al. [21] presents a generic configuration controller composed by only three parts: *monitor, loader* and *configuration store*. In this model, relocation of configurations is not employed. On the other hand, Burns et al. [22] describe the structure of a DRS configuration controller named RAGE (*Reconfigurable Architecture Group*). It has a sub-module specific to relocate configurations, named *Transformation Manager*.

Tab. 2 - Feature comparison of configuration controller models and implementations.

	Models			***Implementations***		
Features	***Shirazi***	***Burns***	***Lysaght***	***Curd***	***Blodget***	***RSCM***
Hardware/Software	NA*	NA	NA	Software	Software	Hardware
Target device	XC6200	XC6200	XC6200	Virtex-II Pro	Virtex-II	Virtex-II
Scheduler type	Static	Static	Dynamic	No	No	Yes
Preemption Support	No	No	Yes	No	No	No
Relocation Support	No	Yes	Yes	No	No	Planned
Configuration storage	No	Yes	Yes	Yes	No	Yes
Configuration decoder	No	No	Yes	No	No	No
Controller Location	NA	NA	NA	FPGA	FPGA	FPGA
Publication date	1998	1997	1999	2003	2003	2004

*NA means Not Applicable

Curd [23] describes a configuration controller implementation for Virtex-II Pro devices. This configuration controller is implemented in software. A code executes in the Virtex-II Pro embedded

PowerPC processor. It reads configuration data from internal RAMs and sends them to the Internal Configuration Access Port (ICAP), an internal port to reconfigure the FPGA available in Virtex-II and Virtex-II Pro series FPGAs. Blodget et al. [24] also implemented a software configuration controller. However, this controller executes on the Xilinx *MicroBlaze* soft processor. Both implementations are based on specific models, different from those previously considered in this Section.

6. RSCM – A CONFIGURATION CONTROLLER

In this Section a generic model of configuration controller called *Reconfigurable System Configuration Manager* or simply RSCM is proposed. Next a feasible implementation based in it is presented and discussed in details. Several characteristics of this model are above presented on last column of the Tab. 2. RSCM aim be a more faithful and complete configuration controller model proposed and implemented, based on current devices and in accordance to existing necessity on partial and dynamic reconfigurable systems deployment.

6.1 RSCM Structure

The *Reconfigurable System Configuration Manager* (RSCM) is a model of configuration controller. The RSCM general structure, detailed in Fig. 2, comprises six modules:

1) Configuration Memory;
2) Self-Configuration Module;
3) Configuration Interface;
4) Reconfiguration Monitor;
5) Configuration Scheduler and
6) Central Configuration Control.

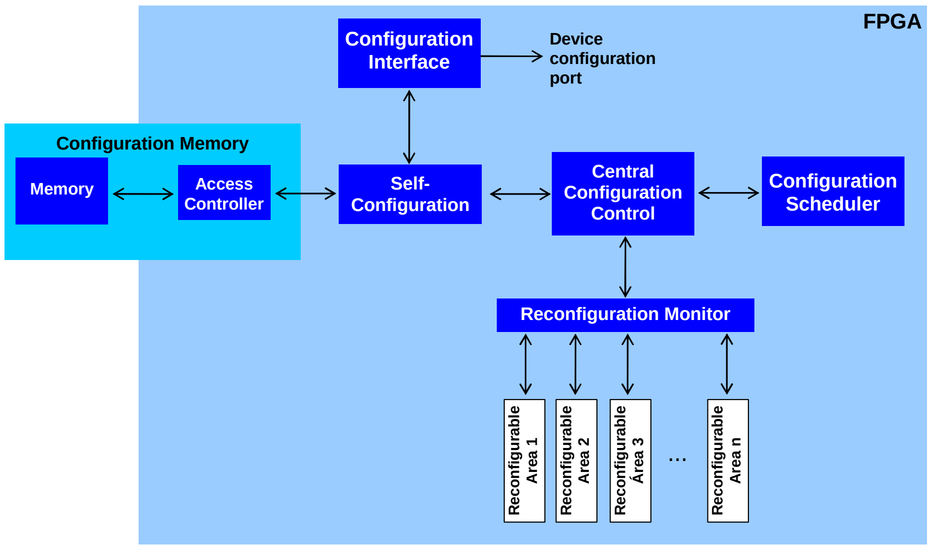


Fig. 2 - RSCM model and its usage in the implementation of DRS.

Configuration Memory (CM)

The *Configuration Memory* stores all partial bitstreams used at runtime by the system. Considering the large amount of memory to store bitstreams, and the scarcity of memory in current FPGAs, the *Configuration Memory* is the only module normally implemented outside the reconfigurable device. How depicted on Fig. 3, support logic to access control to memory and another to initialization is added to this module. Through of it is possible reads and writes configuration data on *CM* using serial interface.

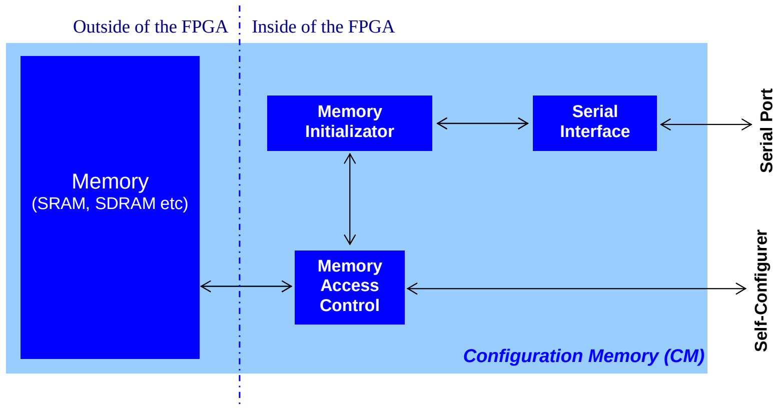


Fig. 3 - Configuration Memory structure. The Memory is implemented outside of the FPGA. The Access Controller and Initialization modules give support to correct operation.

Self-Configuration (SC)

The *Self-Configuration* module controls the configuration process. This module is composed by three main components:

- *Configuration Reader* sends control signals requesting configuration data, and obtains configuration data from *CM.* This data is stored on a buffer, as depicted on Fig. 4.
- *Configurator Machine* reads data from buffer and sends to configuration interface module, which allows sending configuration data to the FPGA.
- *Control Logic* manages previous discussed modules operation. Its interface with *Central Configuration Control* module allows receiving requests to start the configuration process and provides results of the reconfiguration in the form of status signals.

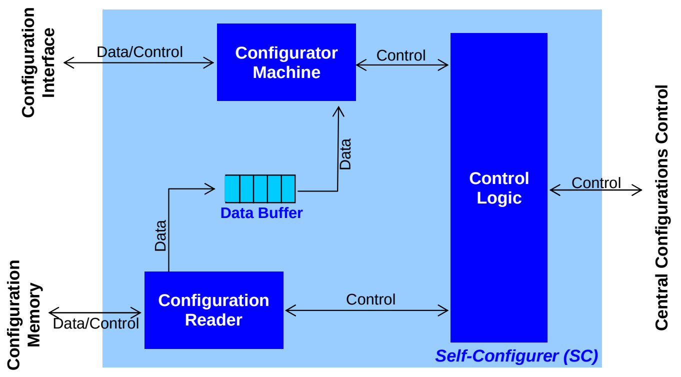


Fig. 4 - Self-Configuration module structure. The data buffer is used to allow concurrent reads from MC and sends to CI configuration data.

Configuration Interface (CI)

The *Configuration Interface* is responsible for receiving configuration data from the *Self-Configuration* module and for sending it to the FPGA configuration port. VirtexII and Virtex II-Pro devices have an internal reconfiguration access port called ICAP (Internal Configuration Access Port) which can be controlled by internal FPGA logic. The ICAP interface is a subset of the SelectMAP interface. Fig. 5 shows a comparison between the two interfaces. The ICAP interface has fewer signals than the SelectMAP interface because it does not have to do full configurations and it does not have to support different configuration modes.

Reconfiguration Monitor (RM)

The *Reconfiguration Monitor* detects situations where reconfigurations need to be performed, the so-called *reconfiguration events*, and notifies the *Central Configuration Control*, which acts appropriately.

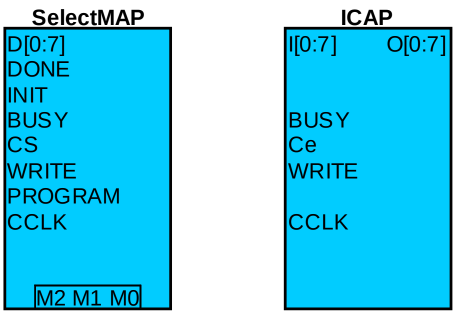


Fig. 5– SelectMAP vs. ICAP interface.

Central Configuration Control (CCC)

The *Central Configuration Control* manages all control flow between other modules of the RSCM system. This module is composed by three main components:

- *Scheduling Requester* requests services to the *Configuration Scheduler* and, stores the bitstream id on a buffer called BSB (*Buffer with Scheduled Bitstreams*);
- *Configuration Requester* reads bitstreams id from BSB and request its reconfiguration process to *Self-Configuration* module according to device allocation status and
- *Control Logic* controls the *Scheduling Requester* and *Configuration Scheduler* modules operation.

Inside of the CCC, an structure called RAT (*Resource Allocation Table*) is used to holds the device allocation status. This information is relevant to define the bitstream device placement.

In summary, *CCC* applies the configuration scheduling stored on the *Configuration Scheduler* module and request reconfiguration performs.

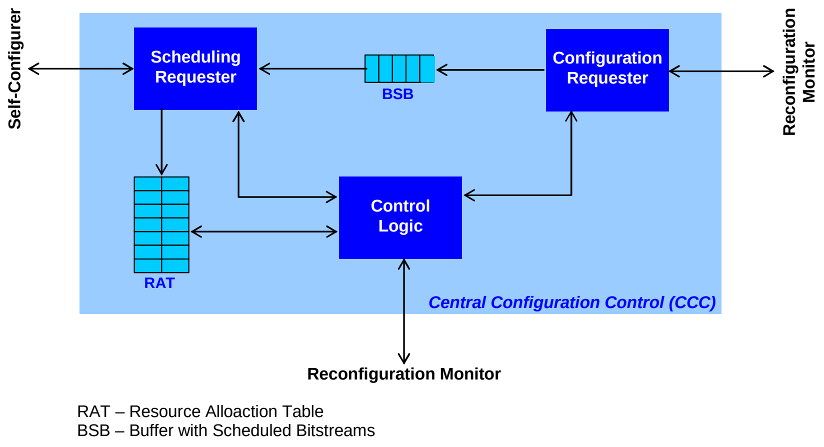
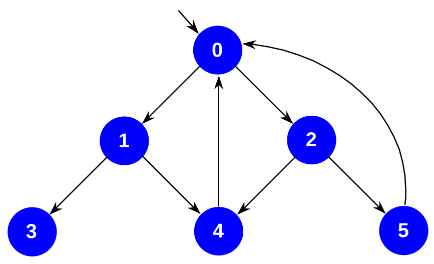


Fig. 6 - Central Configuration Control structure. This module manages the operation of the others RSCM components.

Configuration Scheduler (CS)

The *Configuration Scheduler* module is responsible to determine which configuration is the next to be configured. This module receives service requests from the *CCC*. It stores a data structure with information about configurations dependence, called *Table of Dependencies and Descriptors* (TDD). Fig. 7 shows a TDD example. On the right side a dependence graph is presented and the respective table is presented on left side.

In this work, the RSCM model is implemented in hardware, but since the model is generic, it could as well be implemented in software or mixed hardware/software versions. Since the implemented controller is part of the hardware and lies inside the reconfigurable device containing the rest of the system, the device is capable of performing its own reconfiguration without resource to external controlling devices.

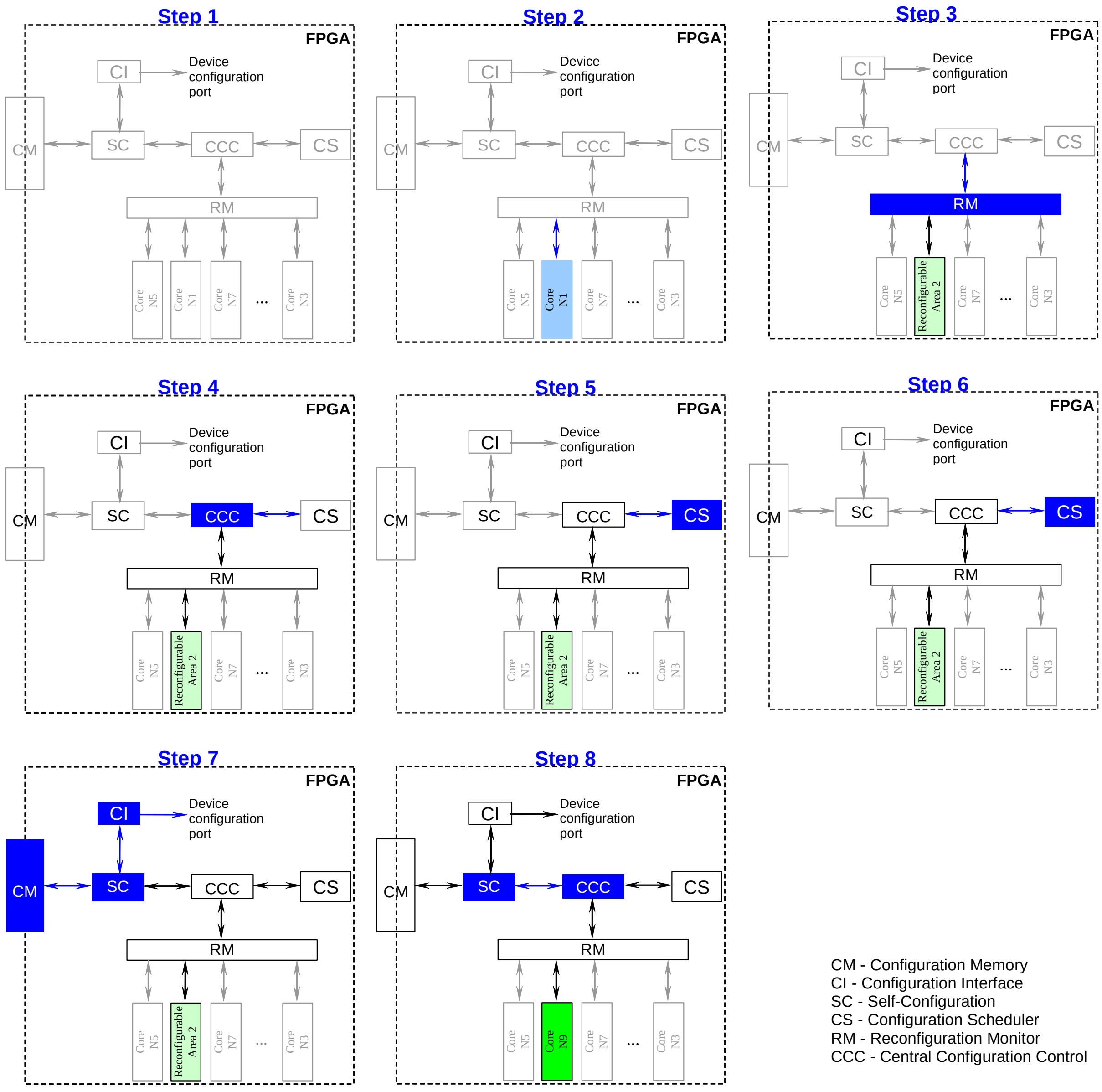


Configs	Npred	Suc 1	Suc 2
0	2	1	2
1	1	3	4
2	1	4	5
3	1	-	-
4	2	0	-
5	1	0	-

Fig. 7 – **Configuration Scheduler structure.** *The table on high side is generated in accordance to dependence graph on left side.*

6.2 RSCM Behavior

To easy comprehend of the RSCM controller the Fig. 8 presents the system behavior during a single reconfiguration event, since of its request detection until its ends. The following eight steps can be notice:



CM - Configuration Memory
CI - Configuration Interface
SC - Self-Configuration
CS - Configuration Scheduler
RM - Reconfiguration Monitor
CCC - Central Configuration Control

Fig. 8 – RSCM operation example. It starts with a reconfiguration condition trigged by the reconfigurable system. Next, each RSCM component performs its tasks to partial an dynamically reconfigure the system, changing the resident bitstreams on device reconfigurable slices according to defined scheduling.

Step 1 - Partial and dynamic reconfiguration system is executing. All reconfigurable slices are allocated to holds bitstreams;
Step 2 - *N1 core* on *slice 2* ends its execution and notifies this event, triggering a *reconfiguration condition*;
Step 3 - *RM* recognizes this condition and signal the *N1* ends performs to *CCC*;
Step 4 - *CCC* receives this signalization, update the RAT status and requests *CS* services;
Step 5 - *CS* searches on the TDD bitstreams ready to perform and sends its codes to *CCC*.
Step 6 - *CCC* store the scheduled bitstreams codes into BSB, searches on RAT free slices and request the reconfiguration of the some bitstream witch its code have stored into BSB.
Step 7 - *SC* read configuration data from *CM* and sends them to *CI*, performing thus a reconfiguration process of the requested bitstream.
Step 8 - *SC* ends its tasks and *CCC* updates the RAT. The reconfigurable system continues its work, with the new bitstream configured on the system.

7. CASE STUDY: R8NR

Possibly, the most intuitive form of DRS is one where the instruction set of a processor is dynamically augmented through the use of reconfigurable hardware. Such components are generically known as *reconfigurable processors*. As a proof of concept for the RSCM controller, this Section proposes a system, named R8NR, composed by a processor with N attached reconfigurable coprocessors. It should be stressed that this implementation allows only a partial validation of the RSCM, since the case study does not encompass functionalities requiring the Reconfiguration Monitor or the Configuration Scheduler. However, the basic functionality of the module, i.e. to automatically control the dynamic substitution of hardware modules of a DRS at runtime was fully validated by the case study.

7.1 The R8NR Project

The R8R processor is based on the R8 processor, a 16-bit load-store 40-instruction RISC-like processor [18]. The original R8 processor was transformed into the R8R processor by the addition of 5 new instructions (Tab. 3) intended to give support to the use of partially reconfigurable coprocessors. The R8R processor was wrapped to provide communication with the local memory, the system bus, the RSCM and the reconfigurable regions. The interface to the reconfigurable regions comprises a set of signals connected to bus macros.

Tab. 3 – Instructions added to the R8 processor in order to produce the R8R processor

Reconfigurable instruction	Semantics description
SELR *address*	Selects the coprocessor identified by *address* for communication with the processor, using the *reconf* signal. If the coprocessor is not currently loaded into the FPGA, the CC automatically reconfigures some area of the FPGA with it.
DISR *address*	Informs the CC, using the *remove* signal, that the coprocessor specified by *address* is dismissed and can be removed if needed.
INITR *address*	Resets the coprocessor specified by *address*, using the *Ioreset* signal. The coprocessor must have been previously configured.
WRR *RS1 RS2*	Sends the data stored in *RS1* and *RS2* to the coprocessor selected by the last *Selr* instruction. *RS1* can be used for passing a command while *RS2* passes data. The coprocessor must have been previously configured.
RDR *RS RT*	Sends the data stored in *RS* to the coprocessor (typically a command or an address). Next, reads data from the coprocessor, storing it into the *RT* register. The coprocessor must have been previously configured.

Fig. 9 displays the organization of the R8NR system implementation. The fixed part in the FPGA is a complete computational system, comprising a simplified version of the RSCM, the R8R processor and its local memory containing instructions and data. Additionally, there is a system bus controlled by an arbiter and peripherals to interface to a host computer, not shown in the picture but implemented. The RSCM acts as a slave of the R8R processor or the host computer. The host computer typically fills or alters the configuration memory before system execution starts.

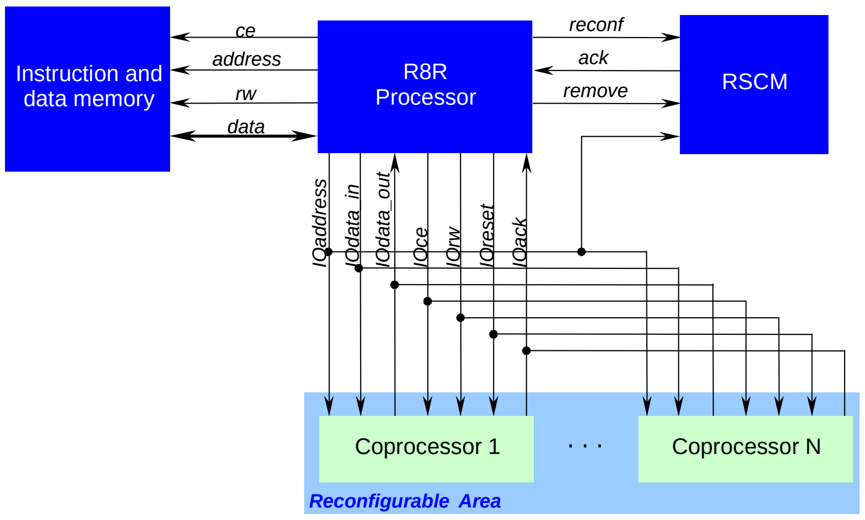


Fig.9 - R8R processor structure. The only coprocessor interface shown is with the R8R processor. Dedicated I/O or memory interfaces are not shown, but allowed for each specific reconfigurable area.

The coprocessors are configured on demand, under control of the software that executes on the R8R processor. During the execution of the system, the R8R selects, at each moment, one specific coprocessor with which it operates. This selection is sent to the RSCM controller that according to the allocation state of reconfigurable areas verifies if the coprocessor is already present in the hardware, reconfiguring some unselected area, if needed. After this, the RSCM notifies the processor that the selected coprocessor is ready. From now on, the software can request coprocessor services. The signal exchange protocol that implements the R8NR inner working is explained next.

The normal operation of the RSCM module is to wait for the R8R processor to produce coprocessor reconfiguration requests using the **reconf** signal, while informing the specific coprocessor identifier in the **IOaddress** lines. If the coprocessor is already in place, the **ack** signal is immediately asserted, which releases the processor to resume instruction execution. If the coprocessor is not configured in some reconfigurable region, the reconfiguration process is triggered. The RSCM is responsible to locate the configuration memory area where lies the coprocessor bitstream corresponding to the identifier. This bitstream is read from memory and sent, word by word, to the Physical Configuration Interface. For Virtex-II devices this corresponds to the ICAP module. In this case, only after the reconfiguration process is over the **ack** signal is asserted. The **remove** signal exists to allow the R8R processor to invalidate some reconfigurable coprocessor. This is useful to help the RSCM to choose the most adequate region to reconfigure next.

For this case study, three coprocessors were implemented. The first, named *SQRT* coprocessor computes the square root of a 32-bit value and presents a 16-bit value as response. The *MULTI* coprocessor executes a multiplication of two 16-bit values and presents a 32-bit response. The *DIV* coprocessor executes a division of two 16-bit values and presents 32 bits response (quotient-remainder).

8. INITIAL RESULTS

The system described in previous Section has been completely prototyped and is operational in two versions, with one (R81R) and two reconfigurable regions (R82R), respectively. A V2MB1000 prototyping platform from Insight-Memec was used. This platform contains a million-gate XC2V1000 Xilinx FPGA, memory and I/O resources.

8.1 Reconfiguration Time

An experiment was conducted to compare the partial and total reconfiguration times, using the *Self-Configuration* module of the RSCM system against the reconfiguration from the download software (*Impact*). Fig. 10 presents a plot comparing the reconfiguration times as a function of the bitstream size (i.e. number of configuration words). It is possible to notice the obviously better performance obtained using the *Self-Configurer* of the RSCM controller.

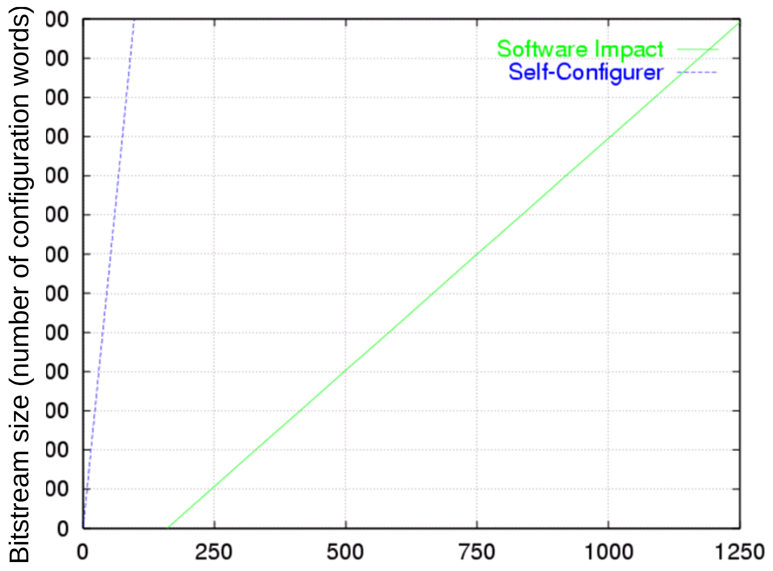


Fig. 10 - Comparing reconfiguration times as a function of bitstream size. A total bitstream to configure a million-gate XC2V1000 Virtex-II device contains 127,581 32-bit words.

8.2 Coprocessors Execution Time

To compare execution times, software implementations of each coprocessor were used. The experiment consisted in comparing the execution time of hardware and software versions versus the number of times the operation is executed. The results are illustrated in Fig. 11.

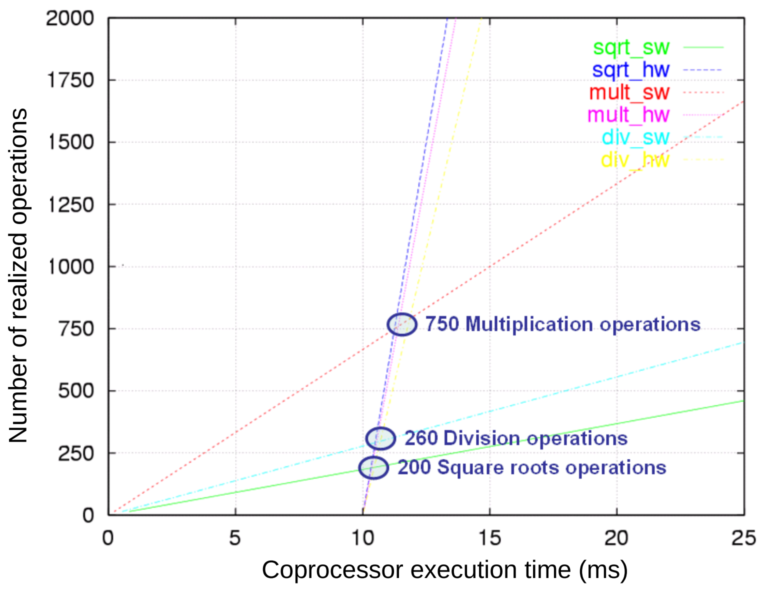


Fig. 11 - Coprocessors execution time versus number of performed operations. The _hw suffix regards hardware implementations, while _sw regards software implementations.

For the multiplication case, the execution of more than 750 consecutive multiplications will execute faster in hardware, even considering the reconfiguration time. For division and square root the respective break-even points occur for 260 and 200 operations. The determination of this break-even point is important to establish the advantage of using DRSs. This break-even point is indeed a worst-case situation, since the use of more than one reconfigurable area, coupled with powerful scheduling strategies implemented by the processor in software can potentially hide reconfiguration times behind the parallel of useful work by other running coprocessors and the processor in itself. All hardware coprocessors were synthesized in the same reconfigurable area, corresponding to bitstreams of exactly the same size. Each bitstream is configured in approximately 10ms by RSCM.

Example applications where the above results can be applied are digital filters where a great number of multiply-accumulate operations are executed over a data set.

8.3 Area Consumption

This Section presents a quantitative analysis for the implemented RSCM controller. These data were obtained from logical synthesis using the Leonardo Spectrum tool. Tab. 4 presents area consumption data for the RSCM controller for a XC2V1000 Virtex-II Xilinx device.

Tab. 4 - Area consumption data for RSCM in a million-gate FPGA. The configuration interface module employs a built-in FPGA module, justifying its null area consumption.

Module	**LUTs**	**DFFs**	**Area (%LUTs)**
Configuration Interface	0	0	0.00 %
Reconfiguration Monitor	16	11	0.16 %
Configuration Scheduler	83	34	0.81 %
Central Configuration Control	126	67	1.23 %
Self-Configuration	268	106	2.62 %
RSCM estimated	493	218	4.81 %

9. Conclusions and Future Work

This work proposed a model and an associated hardware implementation of a configuration controller for DRSs named RSCM. This controller was completely prototyped in hardware and a proof-of-concept reconfigurable processor case study was employed to demonstrate its efficacy.

The RSCM controller presents a small area overhead for medium to large devices (less than 5% of a million-gate FPGA). Execution time quantitative results indicate that the RSCM controller can be used to enable the construction of DRS applications that present performance gains with regard to software only implementations. Much greater improvement on the efficiency of configuration controllers for dynamic reconfiguration can be obtained if e.g. FPGA vendors make these available as optimized standard cells inside their devices. Also, the percentage of area occupied by the configuration controller is further reduced if bigger devices are used.

References

[1] X. Zhang and K. Ng. A review of high-level synthesis for dynamically reconfigurable FPGAs. Microprocessors and Microsystems, v. 24, August. 2000. pp.199-211.

[2] R. Bergamaschi and W. Lee. Designing system-on-chip using cores. In: 37th Design Automation Conference (DAC'00), 2000. pp.420-425.

[3] A. DeHon, Comparing Computing Machines. In: Configurable Computing: Technology and Applications. v. 3526, 1998. pp.124-133.

[4] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, v. 34, no. 2, June 2002. pp. 171-210.

[5] E. Sanchez, M. Sipper, J-O. Haenni, J-L. Beuchat, A. Stauffer and A. Perez-Uribe. Static and

Dynamic Configurable Systems. IEEE Transactions on Computers. v. 48, no. 6, p.556-564. 1999.

[6] P. James-Roxby and S. Guccione. Automated extraction of run-time parameterisable cores from programmable device configurations. In: Field-Programmable Custom Computing Machines (FCCM'00), 2000. pp. 153-161.

[7] Xilinx Inc. The JBits 3.0 SDK for Virtex-II. Available at http://www.xilinx.com/labs/downloads /jbits/index.htm, 2003.

[8] E. Horta, J. Lockwood and S. Kofuji. Using PARBIT to implement Partial Run-time Reconfigurable Systems. In: FPL'02, 2002. pp 182-191.

[9] A. Raghavan and P. Sutton. JPG - A partial bitstream generation tool to support partial reconfiguration in Virtex FPGAs. In: International Parallel and Distributed Processing Symposium (IPDPS'02), 2002. pp. 155-160.

[10] M. Dyer, C. Plessl, M. Platzner. Partially Reconfigurable Cores for Xilinx Virtex. In: FPL'02, 2002. pp. 292-301.

[11] J. Palma, A. Mello, L. Möller, F. Moraes and N. Calazans. Core Communication Interface for FPGAs. In: 15th Symposium on Integrated Circuits and Systems Design (SBCCI'02). Brazil. 2002.

[12] Xilinx Inc. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Application Note XAPP290, Version 1.1. Available at http://www.xilinx.com/xapp/xapp290.pdf, 2003.

[13] E. Caspi, A. DeHon and J. Wawrzynek. A streaming multithreaded model. In: Third Workshop on Media and Stream Processors, 2001.

[14] D. Lehn, R. Hudson and P. Athanas. Framework for architecture-independent run-time reconfigurable applications. In: SPIE’02, November, 2002.

[15] M. Edwards and P. Green. Run-time support for dynamically reconfigurable computing systems. Journal of Systems Architecture, v. 49, 2003. pp.267-281.

[16] T. Bapty, S. Neema, J. Scott, J. Sztipanovits and S. Asaad. Model-integrated tools for the design of dynamically reconfigurable systems, Technical Report #ISIS-99-01, ISIS, Vanderbilt University, 2000.

[17] S. Natarajan, B. Levine, C. Tan, D. Newport and D. Bouldin. Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems. In: MAPLD´99, 1999. pp. 101-107.

[18] M. Eisenring, M. Platzner. A framework for run-time reconfigurable systems, Journal of Supercomputing, v. 21, 2002. pp.145-159.

[19] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul and R.Vemuri. An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. In: RAW’98, Orlando, Florida, USA, March 30, 1998.

[20] D. Robinson and P. Lysaght. Modeling and Synthesis of Configuration Controllers for Dynamically Reconfigurable Logic Systems using the DCS CAD Framework. In: FPL'99, 1999.

[21] N. Shirazi, W. Luk and P. Cheung. Run-Time Management of Dynamically Reconfigurable Designs. In: FPL'98, LNCS v. 1482. Estonia. 1998.

[22] J. Burns, A. Donlin, J. Hogg, S. Singh and M. Wit. A Dynamic Reconfiguration Run-Time

System. In: FCCM'97, 1997. pp.66-75.

[23] D. Curd. Partial Reconfiguration of RocketIO Pre-emphasis and Differential Swing Control Attributes. Application Note XAPP660. Xilinx, 2003.

[24] B. Blodget, S. McMillan and P. Lysaght. A Lightweight Approach for Embedded Reconfiguration of FPGAs. In: 6th Design, Automation and Test in Europe Conference and Exhibition (DATE'03), 2003. pp.399-400.

[25] F. Moraes and N. Calazans. R8 Processor Architecture and Organization Specification and Design Guidelines. Available at http://www.inf.pucrs.br/~gaph/Projects/R8/public /R8_arq_spec_eng.pdf, 2003.