A NoC-based Infrastructure to Enable Dynamic Self Reconfigurable Systems

Leandro Möller[1], Ismael Grehs[2], Ewerson Carvalho[2], Rafael Soares[2], Ney Calazans[2], Fernando Moraes[2]

[1] Darmstadt University of Technology – Institute of Microelectronic Systems
Karlstr. 15, 64283 Darmstadt, Germany
moller@mes.tu-darmstadt.de

[2] Catholic University of Rio Grande do Sul (FACIN-PUCRS)
Av. Ipiranga, 6681 - Prédio 16 - 90619-900 - Porto Alegre - RS - BRASIL
{grehs, ecarvalho, rsoares, calazans, moraes}@inf.pucrs.br

Abstract

Electronic equipments with higher performance, lower power consumption, and smaller size motivate the research for more efficient design methods. Platform-based design is a method to implement complex SoCs that avoids design from scratch. Usually, a platform-based designed SoC includes one or more processors, a real-time operating system, intellectual property (IP) blocks, memories and an interconnection infrastructure. An associated advantage of processor is flexibility at the software level. Hardware is not flexible. Thus, dedicated IP blocks must be inserted at design time. An alternative is to provide the platform with reconfigurable hardware blocks with sufficient capacity to implement any envisaged dedicated IP block. Dynamic self-reconfigurable systems (DSRSs) introduce flexibility to hardware. In DSRSs, IP blocks are loaded according to application demand, an approach that potentially reduces area, power consumption and total system cost.

1. Introduction

Platform-based design [1] is a method to implement complex SoCs, avoiding chip design from scratch. Several IPs other than processors compose SoCs. Examples are communication interfaces, memory controllers and hardware accelerators. These IPs as well as processor may be implemented directly in silicon or using reconfigurable hardware technology. Using the second option, it becomes possible to: (i) improve system performance, by migrating critical tasks to hardware; (ii) build products in smaller devices, thus reducing costs; (iii) extend product life cycle; (iv) update hardware after system manufacturing.

In order to accomplish (i) and (ii), reconfigurable hardware must allow partial and dynamic reconfiguration. Systems using these characteristics are called Dynamically Reconfigurable Systems (DRSs). The main drawback of DRSs is their reconfiguration time. To minimize this drawback, DRSs may be built with the capacity to manage their own reconfiguration process. This can be achieved through the availability of internal reconfiguration ports. Such systems are named Dynamic Self-Reconfigurable Systems (DSRSs) [2]. DSRSs are the target architecture of this work.

One natural implementation choice for DSRSs are dedicated ASICs, with embedded reconfigurable areas. As the goal of this paper is to propose an infrastructure for DSRS, fine-grain reconfigurable FPGAs are used here as a device platform for proof-of-concept purposes. Current FPGAs are clearly limited in terms of useful silicon area, since most of the silicon area is used for programming purposes. In addition, DSRSs may waste a significant amount of this useful silicon to implement the necessary infrastructure. Despite these drawbacks, FPGAs are certainly adequate to prototype the infrastructure proposed herein, serving to demonstrate its benefits, gains and limitations.

An important issue in current SoC design is the implementation of its communication infrastructure. Present SoCs require using scalable communication infrastructures, with shorter wires to minimize power consumption [3]. Networks on chip (NoCs) are an alternative to busses, with several advantages, as stated in [4]. However, few works [5] have suggested mixing reconfigurable IPs and NoCs.

This paper has four goals. First, to propose an infrastructure for DSRSs, identifying which are its required components. The second goal is to present a straightforward design flow supporting DSRSs. The third goal is to describe a NoC actively supporting the process of partial and dynamic IP reconfiguration. The last goal is to depict proof-of-concept case studies, comparing area overhead and reconfiguration time.

The rest of this paper is organized as follows. A discussion about DSRS implementation alternatives is the subject of Section 2. Section 3 presents the Artemis NoC architecture. Section 4 presents a practical design flow to build DSRSs. Section 5 presents and compares two DSRS case studies. Finally, Section 6 presents some conclusions and directions for future work.

2. DSRS Infrastructure

This Section discusses choices and trade-offs associated to the DSRS infrastructure, making a parallel with existent works and recommending implementation choices for each internal component. Figure 1 depicts these components in a DSRS conceptual architecture. The communication infrastructure is presented in Section 3.

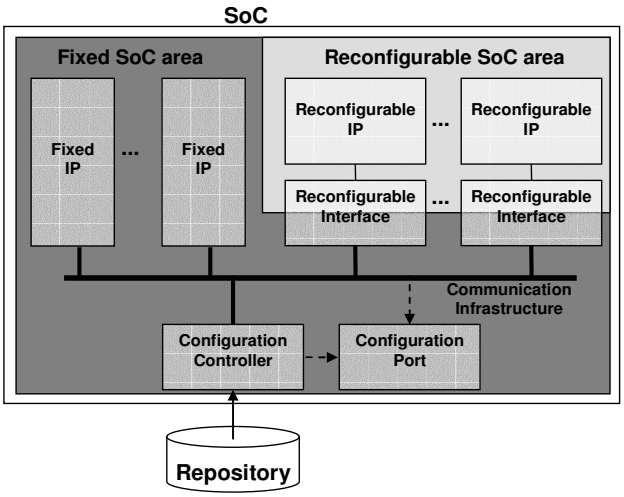


Figure 1 - DSRS conceptual architecture.

2.1. Repositories

DSRSs need to have access to repositories able to store a potentially large number of partial configurations, often called *configuration memory*. Besides stocking partial configurations, these repositories should offer fast access to its contents, to satisfy application requirements. There are basically four device types available to use as configuration memories: (*i*) memory internal to the reconfigurable device, usually available as RAM blocks or BRAMs; (*ii*) devices external to the DSRS using static RAM technology, or SRAMs; (*iii*) devices external to the DSRS using PROM technology, such as EPROM or Flash devices called generically PROMs; (*iv*) devices external to the DSRS using DRAM technology, such as SDRAM.

Applications using BRAMs as repository may support small number of configurations and/or only small configurations, due to its limited capacity. Applications that benefit from difference-based [6] reconfiguration techniques are among those able to employ this kind of repository.

SRAM and DRAM devices present a good compromise between access speed and storage capacity. The former imply simpler controllers added to the DSRSs, but are much more expensive per bit than DRAMs. DRAMs, on the other hand, have a low cost per storage bit, allowing storing more configurations. However, a higher area of the DSRS must be committed to implement its controller.

Contrary to the other three technologies, PROMs have the advantage of keeping configurations after turning the DSRS off. They cost more per bit than DRAMs, but imply a simpler procedure at startup of the DSRS. Also, changing the contents of the repository is more complicated than with the other technologies.

2.2. Reconfigurable Interface

A reconfigurable interface is necessary to implement the communication between a reconfigurable IP and the rest of the DSRS. The interface proposed by Palma et al. [7] uses two levels of tristate buffers in the input and output pins of the reconfigurable IPs. One level of tristates belongs to the reconfigurable IP and the other to the communication infrastructure. Manual routing verification and manual routing corrections are required, to ensure correct connection between IPs. To reduce manual routing, Palma employs a 1-bit data serial bus as communication infrastructure.

Lim and Peattie propose a reconfigurable interface called Bus Macro [6], which employs 8 tristate buffers. Each macro allows the simultaneous exchange of 4 bits between a reconfigurable area and another area, fixed or reconfigurable. The advantage of this macro is that it reduces manual routing. However, it also uses tristate buffers, which are scarce resources in Xilinx FPGAs. The use of such resources overconstrains designs with complex reconfigurable interfaces.

Huebner et al. [8] propose a reconfigurable interface called Bus Macro (distinct from the Xilinx Bus Macro, and herein named Huebner macro). This macro is a static bus used to connect all reconfigurable IPs of the system. This reconfigurable interface is composed by two unidirectional busses, one to communicate the reconfigurable area with the fixed area and another to communicate in the inverse direction. Each macro allows the simultaneous transmission of 8 bits from a reconfigurable area to another area, fixed or reconfigurable.

2.3. Configuration Ports

The external JTAG and SelectMap interfaces are alternatives for implementing configuration ports for DRSs that are not self-reconfigurable, where the configuration controller is located outside the DRS. Although these interfaces can be used for building DSRS (using external wiring connecting some of the reconfigurable device pins to them) most Xilinx devices have available an Interal Configuration Access Port (ICAP). The ICAP usually constitutes the best choice for building DSRSs, since user logic can reach it from inside the reconfigurable device.

2.4. Configuration Controller

The Authors of this paper have built two versions of Configuration Controller (CC): (*i*) a pure hardware version (CC-H); (*ii*) a mostly software version (CC-S). Table 1 compares these two implementations qualitatively.

CC-S is three times slower than the CC-H. This disadvantage is related to the inefficiency of the current API furnished by Xilinx to give access to ICAP. This API requires the CC-S to fetch 512-word blocks of each partial configuration and store these in a BRAM. Only after caching these data, the API sends configuration data to the ICAP. The CC-H sends data directly from an external memory to ICAP, leading to smaller reconfiguration time.

CC-S runs on an embedded 32-bit RISC processor designed by Xilinx, MicroBlaze. The structure of CC-S also includes peripheral device controllers, memory and a communication infrastructure. If configuration control is the only task assigned to this infrastructure, the approach could hardly be justified. However, assuming that most applications today require the use of one or more processors inside the system, and assuming some of these processors have spare time to perform the CC tasks, the additional hardware for configuration control requires less area than CC-H. Given the assumptions above and if the application reconfiguration time requirements are not too stringent, CC-S can be usefully applied.

Table 1 – Comparison of two CC implementations.

Characteristic	CC-H	CC-S
Configuration Speed	Milliseconds	Milliseconds
Area	Requires additional hardware	If processor available, small area overhead (ICAP and macro controllers)
Modification easiness	Complex / extra area	Simple / modifying software

Another important aspect regarding the design of CCs is the easiness for updating/adapting the CC to different applications. When it is necessary to include additional functionalities to the CC, a software implementation is definitely more adequate. Complex tasks can be easily implemented through programming. Examples of such functionalities are configuration compression and on-the-fly decompression, on-the-fly decryption, configuration scheduling policies, and support to configuration preemption. A hardware-only implementation such as CC-H would require restructuring the CC design, realizing the CC re-synthesis and would probably increase the area overhead of the controller.

2.5. DSRS Infrastructure

Table 2 presents some recommended infrastructure choices for DSRSs. Software configuration controllers allow greater flexibility. It is possible to overcome its higher reconfiguration time disadvantage by rewriting the API to access the ICAP module, or by adding a small hardware module to directly manage ICAP.

Table 2 - DSRSs recommended infrastructure.

Infrastructure Element	Recommended Choice
Configuration Controller	Software
Reconfigurable Interface	LUT-Macro
Repository	External SRAM
Reconfigurable Port	ICAP
Communication Infrastructure	NoC

A recommended choice for the reconfigurable interface is to use LUT-macros. Macros developed by Xilinx [6] use a larger area when compared to the LUT-macros proposed in current work (Section 3.2). The Xilinx Bus Macro consumes CLBs from 6 distinct CLB columns, being two in the fixed area and four in the reconfigurable area. Meanwhile, LUT-macros occupy CLBs of only two CLB columns, one at the fixed area and one at the reconfigurable area. Another difference is the number of bits transported by each macro: a Xilinx Bus Macro is 4-bit wide and LUT-macro allows 8-bit wide transfers. CLB columns used for both macros have reduced usability, due to placement and routing restrictions imposed by the macros on both fixed and reconfigurable areas [6].

Another recommendation is to use external static RAM to store partial configurations, since the controller to access these memories is very simple, present a small access time, and the capacity of such memories is sufficient to store several partial configurations. It is not advisable to waste internal FPGA memory with partial configurations, since the capacity of such memories is too small.

3. Artemis NoC

The last component of the proposed DSRS infrastructure discussed here is the communication infrastructure. As stated before, NoCs are good choices due to their scalability, increased parallelism and short-range wires that reduce power consumption. This work proposes Artemis, a NoC that supports specific reconfiguration services and is based in the Hermes NoC [9]. This Section describes the modifications carried out in Hermes to allow its use in DSRSs.

The partial reconfiguration process may produce glitches in the interface between the IP under reconfiguration and the rest of the device. These glitches may introduce spurious data into the NoC, causing malfunctions or even circuit blocking. In addition, packets transmitted to an area suffering reconfiguration, must be discarded, since it is typically impossible to know if these packets are targeted to the previous configuration in this area or to the next reconfiguration. To avoid such problems, a set of *services* must be added to the NoC to enable its use in DSRSs.

Three services are implemented in Artemis: (*i*) reconfigurable area insulation; (*ii*) packet discarding; (*iii*) reconfigurable area reconnection. Hermes passed through the addition of two functionalities to support these services: (*i*) definition of control packets, enabling IPs to send packets to routers, not only to other IPs; (*ii*) capacity to disconnect/connect routers from its associated reconfigurable area. These functionalities are detailed in the next Sections.

3.1. Control packets: structure and function

The addition of two sideband signals per port to the original Hermes router serves to differentiate control packets from data packets. These signals, depicted in Figure 2, are *ctrl_in* and *ctrl_out*. For each flit sent by *data_out*, the *ctrl_out* is asserted together with *tx* if the flit is a control packet. The target router receives flits

analogously, using *data_in*, *rx* and *ctrl_in* signals.

When the reconfigurable area is insulated, the router discards any data packets sent to the area under reconfiguration. Insulation also protects the network, since during reconfiguration transients can occur in the reconfigurable interface. If such signals are considered, spurious data may enter the NoC. Transients were indeed observed in hardware by measuring the router-IP interface with a logic analyzer during reconfiguration. These events may signal a *false packet* to the router, with unpredictable outcomes. Once the new IP is configured, a control packet reconnects IP and router, enabling normal operation.

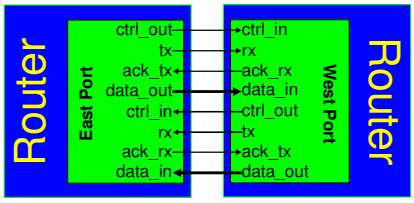


Figure 2 – Interface between Artemis routers.

The reception and forwarding of control and data packets are similar. The major change in the router is the addition of one bit at each position of the input buffer. This is required to propagate the value of the *ctrl_out* signal to the reconfigurable IP router. When the control packet arrives at its destination router, it decodes and executes the corresponding operation.

3.2. Reconfigurable IP to router interface

This work proposes a new reconfigurable interface that does not impose the use of a specific communication infrastructure. This interface uses LUTs. Two unidirectional macros compose the reconfigurable interface, as depicted in Figure 3. The first one, named F2R, is responsible to send data from the fixed part of the system to a reconfigurable IP, while the second one, named R2F, implements the communication in the inverse direction. Both macros allow the simultaneous transmission of 8 data bits. The F2R macro is an identity function, while the R2F uses a special logic to avoid transient glitches during the reconfiguration process from reconfigurable to fixed areas.

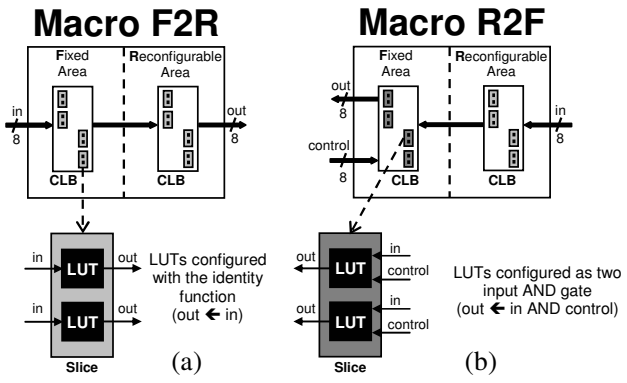


Figure 3 – Proposed macros: (a) F2R; (b) R2F.

The complete interface between the Artemis router and a reconfigurable IP appears in Figure 4. It uses two R2F macros to connect 10 bits from right to left and two F2R macros to connect 11 bits in the inverse direction. The interface between the router and the reconfigurable IP does not contain the *ctrl_in* and *ctrl_out* signals because reconfigurable IPs neither send nor receive control packets. The *reset* is a global signal used to initialize the entire system. The router asserts the *reconf* signal to initialize the reconfigurable core connected to the local port. The *reconf_n* signal in Figure 4 connects to the *control* signal in Figure 3, controlling the connection from the router to the reconfigurable core.

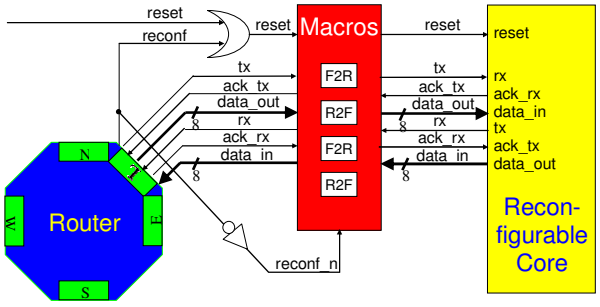


Figure 4 – Router to reconfigurable core interface.

4. Design Flow for DRS

The layout of reconfigurable IPs shares some properties: (*i*) logic of a reconfigurable region must lie inside it (achieved with placement restrictions); (*ii*) wires of a reconfigurable region must lie inside it (achieved with routing restrictions); (*iii*) fixed communication interface with the rest of the DRS. Next Sections details the main design flow steps to implement a DRS/DSRS.

4.1. Reconfigurable interfaces insertion

To enable the use of reconfigurable IPs, it is necessary to impose two restrictions in reconfigurable interfaces: reconfigurable IPs sharing the same region must present identical interfaces (in terms of number and type of signals) and identical placement of interface pins. One way to define reconfigurable interface pins is to insert pre-defined feedthrough components, named *macros*. Figure 5(a) illustrates a system with one fixed IP, two reconfigurable IPs and *macros* defining the interface pins. *Macros* are inserted in the system description (e.g. VHDL or Verilog).

4.2. Placement constraints

The second step is to constrain the placement of IPs and *macros*, as presented in Figure 5(b). A floorplanner tool may constrain the placement and shape of the system IPs (fixed and reconfigurable IPs), as well as the placement of *macros*. Standard place and route follows the constraints insertion.

4.3. Routing verification / modification

In the current generation of Xilinx physical synthesis tools, floorplanning restrictions do not have influence on the routing tool. As illustrated in Figure 5(b), some wires can still cross reconfigurable region boundaries. If this situation occurs, the associated signal can be disconnected after a reconfiguration step, possibly causing a system malfunction. This situation pervades all reconfigurable design flows, including Xilinx Modular Design. In this case, the designer must either reroute the wire(s) crossing the interfaces (manually or automatically) or go back to the previous step, to try different placement constraints. The final routing must be similar to the one presented in Figure 5(c), where no wire crosses a reconfigurable interface. One noticeable exception to this rule is the global clock signals, which can safely cross the whole chip.

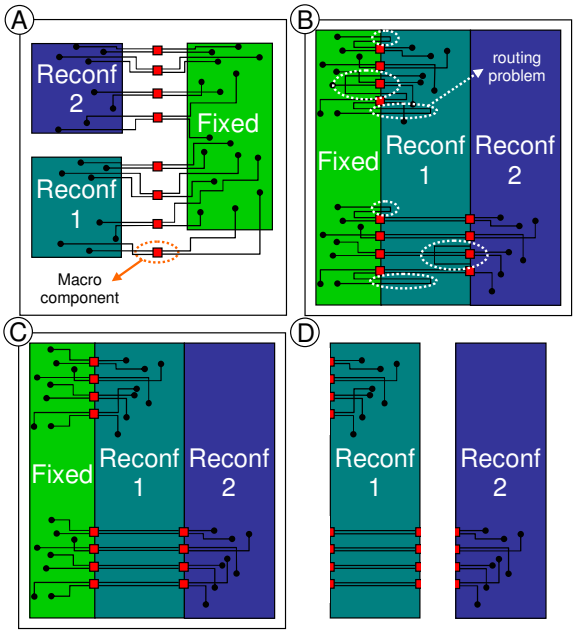


Figure 5 – DRS flow proposed in this work.

4.4. Partial configurations generation

Partial configurations, or partial *bitstreams*, are a set of bits used to configure a DRS. Partial bitstream generation is done by extracting a section of a total bitstream, corresponding to a reconfigurable region. This is illustrated in Figure 5(d). It is important to include part of the macro component in partial bitstreams to connect the reconfigurable core to the fixed part of the DRS. The method used here to generate partial bitstreams is straightforward, a one-phase flow. Assignment of another core to the same region requires partially repeating the flow for each core, while keeping the same placement constraints. Two tools may generate partial bitstreams. The first one is the proprietary Xilinx tool, *BitGen*, with specific commands to define the coordinates of the reconfigurable core. The second tool, compatible with all Virtex-II (Pro) devices, was developed by the authors.

4.5. Core relocation

Two situations require to partially repeating the DRS flow. The first one arrives with the assignment of different cores to the same reconfigurable region. The second one arrives with the assignment of the same core to different reconfigurable regions. It is possible to avoid the second situation if the same bitstream can be loaded at different regions. This procedure is named *relocation* [10]. A core originally synthesized for one reconfigurable region can be moved to another one, without re-synthesis. Core relocation also reduces the memory requirements to store partial bitstreams, diminishing system cost.

5. Case Studies

This Section presents the implementation of two proof-of-concept DSRS case studies and their comparison. Table 3 details the characteristics of the OPB-based (Figure 6) and Artemis-based (Figure 7) case studies. These case studies allow DSRS design space exploration, evaluating benefits, gains and limitations of each infrastructure element.

Table 3 - Case studies implementation characteristic

Infrastructure Element	OPB-based DSRS	Artemis-based DSRS
Configuration Controller	Software (CC-S)	Hardware (CC-H)
Reconfigurable Interface	LUT-Macro	LUT-Macro
Repository	Internal BRAM	External SRAM
Reconfigurable Port	ICAP + Xilinx API	ICAP + dedicated hardware
Communication Infrastructure	OPB Bus	Artemis NoC

5.1. OPB-based DSRS Description

The OPB-based DSRS contains a Microblaze processor, running an application and the configuration controller (CC-S). The system also contains several IPs connected to the OPB bus, as shown in Figure 6.

The design flow to synthesize this DSRS requires additional steps w.r.t. the one presented in Section 4. A similar flow is also used in [11]. The steps to build the OPB-based DSRS are:

- Build an initial system, using the Embedded Development Kit (EDK) with the Xilinx IPs and the reconfigurable IP (user function + macros + OPB wrapper);
- Insert macros to insulate the user function from the fixed part (Section 4.1). These macros are located between the IPIF interface and the user function (the user module template generated by EDK offers to the user an interface simpler than the OPB bus, named IPIF). Even if IPIF is simpler than OPB, it has 80

signals (36 from left to right, 44 from right to left), requiring 11 macros (5 R2F macros, 6 F2R macros), complicating floorplaning and routing steps;

- Generate the system netlist with EDK, exporting it to ISE (Integrated Software Environment);
- Execute the logic synthesis, followed by floorplanning (Section 4.2) and physical synthesis (Section 4.3). The result of this step is the complete bitstream of the SoC;
- Import results back to EDK for software generation. The binary code is finally added to the complete bitstream.

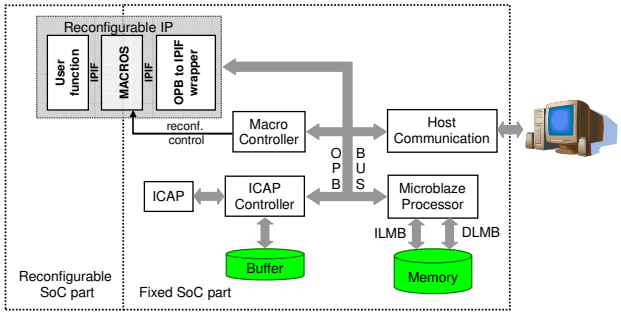


Figure 6 – The OPB-based DSRS structure.

The above steps are repeated for each reconfigurable IP. Partial bitstreams (Section 4.4) are extracted from the obtained complete bitstreams. The OPB-based DSRS was prototyped in a Memec Insight platform with a Virtex-II Pro XC2VP30 device.

OPB-based DSRSs have two drawbacks: bus-based communication and limited internal repository. Additionally, the design flow is quite complex, since two software environments are used: EDK and ISE. However, this simple case study allows reconfiguration time evaluation using the Xilinx API to access the ICAP module, and the area consumed to implement the reconfiguration infrastructure.

5.2. Artemis-based DSRS Description

The Artemis-based DSRS contains a 2x2 NoC used as communication infrastructure and several IPs as illustrated in Figure 7. The MR2 processor is a 32-bit RISC processor, based in a load-store MIPS architecture, with 27 distinct instructions, a 32x32 register file, non-pipelined. The processor uses four internal 18 Kbits RAM blocks as instruction and data memories, providing 1K words in each memory. Three different arithmetic IP modules can be used as reconfigurable IPs: "mult" (multiplies two 16-bit operators), "div" (divides one 16-bit operator by a 16-bit operator) and "sqrt" (extracts the square root of a 32-bit operator).

The processor is the system master. Memory mapped instructions access reconfigurable IPs. The following system operating protocol is used:

- the processor sends a packet to the CC, informing the identification of the desired IP.
- the CC (i) receives the reconfiguration request; (ii) selects a reconfigurable area where to configure the requested IP (if more than one reconfigurable area is available); (iii) sends a packet to disconnect communication between the router and the selected reconfigurable area; (iv) read the specific bitstream, transmitting it to ICAP.
- After reconfiguration, the CC sends a packet to reconnect communication between router and the configured IP. A second packet is sent to the processor with the network address where the IP was configured.

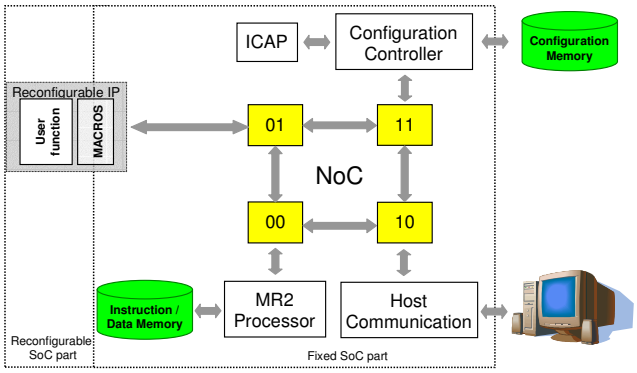


Figure 7 – Artemis-based DSRS.

The Artemis-based DSRS was also prototyped in a Memec Insight platform with a Virtex-II Pro XC2VP30 device. The design flow used to synthesize this DSRS employs the straightforward flow presented in Section 4. This is simpler than the flow used for the OPB-based DSRS, since only the ISE environment needs to be used.

Except for the configuration controller, this DSRS follows the recommended choices to implement DSRS. The configuration controller is implemented in hardware, favoring performance, but reducing flexibility.

5.3. Infrastructure comparison

A common choice for both experiments presented is the use of LUT macros. LUT macros were employed in the OPB-based DSRS due to the number of bits in the reconfigurable interface (80), therefore reducing the number of CLB rows when compared to Xilinx Bus Macros. The LUT macros had to be extended to occupy 4 CLB columns each to achieve successful interface routing. The Artemis-based DSRS has a less complex interface (21 bits), using four LUT macros, exactly as presented in Figure 4, and occupying only 2 CLB columns each.

A second common choice in both experiments is the ICAP configuration port. The first case study uses the Xilinx API to access the ICAP port, while the second case study uses a dedicated module developed to access the ICAP port. As already mentioned, the Xilinx API is slower than dedicated hardware due to current buffering requirements. Table 4 compares the partial bitstream sizes and reconfiguration times.

The third column presents partial bitstream sizes. Partial bitstreams of the OPB-based DSRS occupy 10 CLB columns, while for the Artemis-based DSRS they occupy 6

CLB columns[1]. It is possible to store partial bitstreams of the OPB-based DSRS in internal BRAMs because a simple compression algorithm was applied to partial bitstreams, based on zeroes/ones counting. On-the-fly software decompression is executed before sending bitstreams to the ICAP controller. There is no time penalty in this decompression, due to the algorithm simplicity. The Artemis-based DSRS stores partial bitstreams in a 1 Mbyte external SRAM. The Artemis-based DSRS stores up to 10 partial bitstreams, without compression, while the OPB-based DSRS is able to store only 2 partial bitstreams using compression.

Table 4 – Reconfiguration times† for OPB and Artemis based DSRS case studies.

Case Study	Partial Bitstream Size		Reconf. Time		Minimal Reconf. Time
	Module Name	Size (Kbytes)	CC-H	CC-S	
OPB-based	Arith. 1 / 2	182,180	-	63.55	3.64
Artemis-based	Multiply	99,644	9,98	34.76*	1.99
	Divider	96,428	**9,65**	33.63*	1.93
	Square Root	101,988	10,21	35.57*	2.04

†Times are expressed in milliseconds and reconfigurations run at 50MHz.

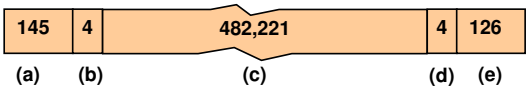
*Estimated, using data from the OPB-based system.

The fourth and fifth columns present the reconfiguration time using the CC-H and CC-S configuration controllers. The CC-H reconfiguration time is in average three times faster than CC-S, considering the NoC protocol. Reconfiguration times were measured using two methods: internal FPGA timers and a logic analyzer.

The sixth column presents the minimal reconfiguration time, assuming it would be possible to transmit one partial bitstream byte per clock cycle (at 50 MHz). This column shows that it is not possible to work with reconfiguration times below 1 ms in current case studies, with reconfigurable IPs using 6 to 10 CLB columns. With more complex reconfigurable IPs, reconfigurable area is expected to increase consequently increasing the reconfiguration time.

Figure 8 details the reconfiguration time for the *divider IP*. The reconfiguration time, 9.65 ms, is equivalent to 482,500 clock cycles. Observe that 99.94 % of this reconfiguration time is spent by the reconfiguration process itself (Figure 8(c)), with a very small time spent in the NoC with control packets.

After reconfiguration, the protocol to access the reconfigurable IP comprises three steps: (*i*) creation and transmission of a packet with the operators to the reconfigurable IP; (*ii*) creation and transmission of a read packet to receive results; (*iii*) reception of the result packet from the reconfigurable IP. Typical time spent in each step is 173, 141 and 117 clock cycles respectively. As the reconfigurable IPs are very simple in this case study, once the read request arrives at the reconfigurable IP, the packet with the results is sent immediately to the source IP, totalizing in average 439 clock cycles (sum of the time spent in each step). This protocol can be simplified by eliminating the read packet (141 cycles), sending the answer from the reconfigurable IP directly to the source IP.



(a) packet from a source IP to the CC asking a new reconfigurable IP
(b) CC processing time and packet to the reconfigurable area to disconnect it
(c) reconfiguration time
(d) packet from the CC to the new reconfigurable IP reconnecting it
(e) packet from the CC to the source IP with the reconfigurable IP address

Figure 8 – Reconfiguration protocol timing, in clock cycles, for Artemis-based DSRS.

At 50 MHz, 10 ms represent 500,000 clock cycles. This reconfiguration time can be hidden by: (*i*) executing complex computations in hardware; (*ii*) pre-fetching reconfigurable IPs to later use; (*iii*) reusing the same reconfigurable IPs during a time longer than the execution in software plus the time to configure the IP into the DSRS. With such strategies, the reconfiguration time has minimal impact in DSRS performance. For example, if a given function executed in hardware is 500 clock cycles faster than an equivalent software implementation, after 1,000 consecutive executions the hardware implementation displays superior performance. This can be easily achieved with image processing algorithms, where the same operation is repeated thousands of times.

For these proof-of-concept case studies, the average execution time for the equivalent software implementation is 26% slower (in average 600 clock cycles against 439 clock cycles). This difference in favor of the hardware implementation, 161 cycles, is not yet sufficient to demonstrate performance gains for the proposed infrastructure, but clearly shows its viability. Some application portions (typically loops) may benefit from this approach, given they consume at least 1,000 clock cycles in the embedded processor and are repeatedly used.

Table 5 and Table 6 compare the area to implement both DSRSs. The first analysis concerns the configuration controller (CC) area overhead. The CC-H uses 494 slices. The CC-S uses 821 slices (Microblaze, ICAP and macro controllers). However, if a processor is already available in the system (such as MicroBlaze), the area of the CC-S represents the area of the ICAP and macro controllers, resulting in 250 slices. As processors are ubiquitous in actual SoCs, a software CC represents the implementation option with smaller area overhead.

The area of the Artemis-NoC is 1167 slices (Table 5), representing in average 290 slices per router. For this case study, this area represents an important overhead. In practice, when using real IPs, an area overhead of 5-10% per IP is expected, justifying the use of NoCs in DSRSs. Comparing the router area to the Gecko platform [5], Gecko routers consume 611 slices (router plus network interfaces, data and control).

[1] Different bitstream sizes for the same number of CLB columns exists because partial bitstreams are generated by *Bitgen*, which uses the multi-frame write feature.

Table 5 - Artemis-based DSRS area report (XC2VP30)

IP	# Slices (total: 13696)		# FF (total: 27392)	
	Total	Percentage	Total	Percentage
Serial	316	2.31%	279	1.02%
Processor	1001	7.31%	555	2.03%
CC (CC)	**494**	**3.61%**	**294**	**1.07%**
Artemis NoC	1167	8.52%	959	3.50%
DIV (reconf IP)	183	1.34%	259	0.95%
MULT (reconf IP)	172	1.26%	259	0.95%
SQRT (reconf IP)	223	1.63%	269	0.98%

Table 6 - OPB-based DSRS area report (XC2VP30).

IP	# Slices (total: 13696)		# FF (total: 27392)	
	Total	Percentage	Total	Percentage
MicroBlaze	571	4.17	366	1.34
MicroBlaze Perip.	160	1.17	75	0.27
MicroBlaze OPB	90	0.66	11	0.04
ICAP Controller	**151**	**1.10**	**155**	**0.57**
Macro Controller	**99**	**0.72**	**136**	**0.50**
Arith1 (reconf IP)	128	0.93	168	0.61
Arith2 (reconf IP)	128	0.93	168	0.61

6. Conclusion and Future Work

The main contribution of this work is the proposal of a conceptual DSRS architecture, summarized in Table 2, centered on the use of a NoC interconnection. The implementation of two proof-of-concept case studies demonstrates the viability of the proposed DSRS architecture, even if none of the case studies follow all recommendations. However, each recommendation in the Table was implemented and evaluated by at least one of the case studies. To support the development of the proposed DSRS architecture, the paper advanced two additional contributions: (*i*) a suggestion of a straightforward DSRS design flow; (*ii*) the design of a specific NoC supporting partial and dynamic hardware reconfiguration.

The ideal implementation choice for this DSRS architecture is dedicated ASICs with embedded reconfigurable areas. Nonetheless, partial and dynamic reconfigurable FPGAs were used to successfully prototype the architecture. The main advantage of the suggested flow is a reduced number of steps compared to other flows proposed in the literature, such as Modular Design. The proposed flow employs new macros, which guarantee the correct operation of the rest of the system during reconfiguration, avoiding the use of tristate buffers, components scarcely available in Virtex FPGAs. Also, the new macros enable the use of communication architectures other than busses to link reconfigurable modules to other parts of the system. To support dynamic IPs reconfiguration, the paper showed the need to add services to ordinary NoCs. Three needed services were identified: IP insulation, packets discarding and IP reconnection. These services were implemented over the existing Hermes NoC, resulting in the Artemis NoC, which supports DSRS.

The case studies evaluation helped to identify the area overhead incurred by the proposed infrastructure and the reconfiguration time. The addition of a Configuration Controller in a SoC represents a small area overhead (1.82 to 3.61% of the available slices for XC2VP30 device), while providing a greater flexibility to the system. The addition of hardware flexibility to a SoC enables to implement the same function both in software and in hardware. The user or the operating system may select the implementation according to performance requirements. The experiments allowed to observe that, independently of the fact that reconfiguration is controlled in software or hardware, IP reconfiguration time is always above 2 ms for current FPGA technologies (measured times were between 9.65 ms and 63.55 ms). This represents an average value of 500,000 clock cycles. The time measured to send data to the reconfigurable IP, and to receive data from it, through the NoC is around 439 clock cycles. Performance gains can be easily obtained in loops with small/medium complexity (1,000 clock cycles) or more complex IPs.

7. References

[1] Keutzer, K.; Newton, A.R.; Rabaey, J.M.; Sangiovanni-Vincentelli, A. "System-Level Design: Orthogonalization of Concerns and Platform-Based Design". IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 19 (12), Dec. 2000, pp. 1523-1543.

[2] Van den Branden, G.; Touhafi, A.; Dirkx, E. "A design methodology to generate dynamically self-reconfigurable SoCs for Virtex-II FPGAs". In: FPT'05, 2005, pp. 325-326.

[3] Dally, W.; Towles, B. "Route Packets, Not Wires: On-Chip Interconnection Networks". In: DAC'01, 2001, pp. 684-689.

[4] Benini, L.; De Micheli, G. "Networks on Chips: a New SoC Paradigm". Computer, vol. 35 (1), Jan. 2002, pp. 70-78.

[5] Marescaux, T.; Nollet, V.; Mignolet, J.-Y.; Bartic, A.; Moffat, W.; Avasare, P.; Coene, P.; Verkest, D.; Vernalde, S.; Lauwereins, R. "Run-Time Support for Heterogeneous Multitasking on Reconfigurable SoCs". Integration, the VLSI Journal, vol. 38 (1), Oct. 2004, pp. 107-130.

[6] Lim, D.; Peattie, M. "Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations". Xilinx Application Note 290 (v1.0), 2002.

[7] Palma, J.; Mello, A.; Möller, L.; Moraes, F.; Calazans, N. "Core Communication Interface for FPGAs". In: SBCCI'02, 2002, pp. 183-188.

[8] Huebner, M.; Paulsson, K.; Becker, J. "Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores". In: IPDPS'05, 2005, pp. 149a-149a.

[9] Moraes, F.; Calazans, N.; Mello, A.; Möller; Ost, L. "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Integration, the VLSI Journal, vol. 38 (1), Oct. 2004, pp. 69-93.

[10] Krasteva, Y.; Jimeno, A.; Torre, E.; Riesgo, T. "Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in FPGAs". In: RSP'05, 2005, pp. 77-83.

[11] Donato, A.; Ferrandi, F.; Santambrogio, M.D.; Sciuto, D. "Caronte: a complete methodology for the implementation of partially dynamically self-reconfiguring systems on FPGA platforms". In: FCCM'05, 2005, pp. 321-322.