Infrastructure for Dynamic Reconfigurable Systems: Choices and Trade-offs

Leandro Möller, Rafael Soares, Ewerson Carvalho, Ismael Grehs, Ney Calazans, Fernando Moraes
{moller, rsoares, ecarvalho, grehs, calazans, moraes}@inf.pucrs.br

Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS)
Av. Ipiranga, 6681 - 90619-900 - Porto Alegre - BRASIL

ABSTRACT
Platform-based design is a method to implement complex SoCs, avoiding chip design from scratch. A promising evolution of platform-based design are MPSoC. Such generic architectures might furnish enough performance for several classes of embedded systems. An associated advantage of these architectures is flexibility at the software level. In principle, hardware is not flexible. Thus, dedicated IP blocks must be inserted before chip design, or enough area can be reserved for them when using reconfigurable blocks. Dynamic self-reconfigurable systems (DSRSs) introduce flexibility to hardware. In DSRSs, IP blocks are loaded according to application demand, reducing area, power consumption and system cost. An MPSoC based platform, associated with dynamic reconfiguration, provides both hardware and software flexibility. This paper has two main goals. First, to present the necessary infrastructure for DSRSs, identifying which components are required in these systems, such as a configuration controller, configuration ports and reconfigurable IP interfaces. The second objective is to discuss practical implementations choices and area-performance tradeoffs. The paper employs case studies to access the advantages and problems related to different implementations for the communication infrastructure (bus and NoC), the configuration controller (hardware and software) and IP interfaces (LUT and tristate based).

Categories and Subject Descriptors
B.7.1 [Integrated Circuits]: Types and Design Styles – advanced technologies, algorithms implemented in hardware, VLSI (very large scale integration).

General Terms
Design, Experimentation, Measurement, Performance, Theory, Verification.

Keywords
Reconfigurable Architectures, Configuration Controllers, NoCs.

SBCCI'06, August 28–September 1, 2006, Minas Gerais, Brazil.

1. INTRODUCTION
Platform-based design [1] is a method to implement complex SoCs, avoiding chip design from scratch. Usually, a platform-based SoC includes one or more processors, a real-time operating system, intellectual property (IP) blocks, memories and an interconnection infrastructure. Depending on the platform type, it is possible to customize it by adding hardware IPs, including reconfigurable hardware blocks like FPGAs and/or writing embedded software.

A promising evolution of platform-based design are MPSoC based platforms, with systems composed by a set of processors interconnected by a communication infrastructure. An advantage of these architectures is flexibility at the software level. In principle, hardware is not flexible. Thus, dedicated IP blocks must be inserted before chip design, or enough area can be reserved for them when using reconfigurable blocks.

Systems with multiple processors in a system on chip (MPSoC) present as main characteristic software flexibility and the performance of distributed systems. On the one hand using *homogeneous* processors in MPSoCs guarantees task migration facilities, on the other hand *heterogeneous* processors can provide flexibility to the MPSoC to choose the most adaptable processor to execute a certain task. One new degree of flexibility on heterogeneous MPSoCs can be obtained by using reconfigurable devices. These devices allow: (*i*) extending product life cycle; (*ii*) updating hardware; (*iii*) building smaller products; (*iv*) migrating tasks to hardware.

In order to accomplish (*iii*) and (*iv*) the reconfigurable device must have the characteristic to be partially and dynamically reconfigured. Systems that use this characteristic are called Dynamically Reconfigurable Systems (DRSs). Another characteristic that can speed up the reconfiguration at runtime is if the reconfigurable device itself manages the hardware substitution. In this case the system can be named Dynamically Self Reconfigurable System (DSRS).

This paper has two main goals. First, to present the necessary infrastructure for DSRSs, identifying which components are required in these systems, such as a configuration controller, configuration ports and reconfigurable IP interfaces. The second objective is to discuss practical implementations choices and area-performance tradeoffs.

The rest of the paper is organized as follows. Section 2 presents concepts related to reconfigurable systems. Section 3 presents

related works in hardware infrastructure for DSRS, while Section 4 discusses choices and trade-offs among some of these works. A comparison between two self reconfigurable infrastructures is the subject of Section 5. Finally, Section 6 presents some conclusions and future work.

2. CONCEPTUAL RECONFIGURABLE ARCHITECTURE

The most widespread commercial ICs that use reconfigurable hardware are the RAM-based FPGAs. These FPGAs can be configured an unlimited number of times by storing a valid configuration in its *configuration memory*. The process of writing the configuration memory programs the FPGA resources (logic functions, I/Os and interconnection points). A *valid configuration* is a sequence of values that defines a coherent *virtual structural* behavior of the hardware. The term *configuration* is also used to define the process of writing the configuration memory, while the term *reconfiguration* defines the process of rewriting the FPGA configuration memory with another valid configuration.

Totally reconfigurable systems are systems that write the complete device configuration memory in each reconfiguration step, in contrast to the *partially reconfigurable systems*. *Statically reconfigurable systems* are those that suspend the system operation during the reconfiguration process, in contrast to the *dynamically reconfigurable systems* (DRSs), which allow that part of the system continues to operate throughout the reconfiguration process. DRSs able to control their own reconfiguration process are called *dynamically self reconfigurable systems* (DSRSs). These systems must contain specific resources to control new charges of configurations at runtime, as depicted in Figure 1.

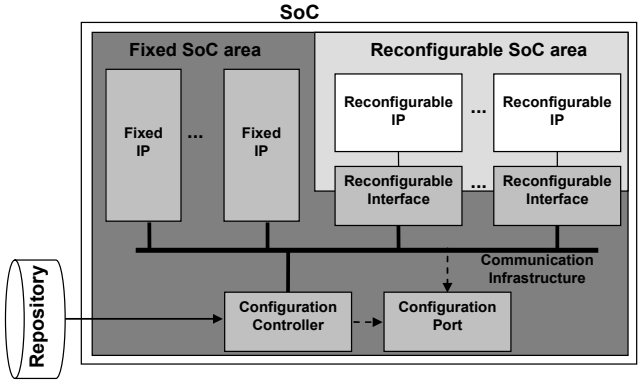


Figure 1 – DSRS generic architecture.

Besides a set of fixed IPs (not reconfigurable) and the communication medium, which are IPs that usually compose SoC designs, there are a set of reconfigurable IPs and four other fixed IPs: a Configuration Controller (CC), a Repository, a Configuration Port (CP) and one Reconfigurable Interface (RI) associated to each reconfigurable IP. The CC loads configurations stored in a repository and, according to its scheduler, chooses one reconfigurable IP to be substituted. Then, the configuration of a reconfigurable IP is sent to the Configuration Port (CP) and the system starts to be immediately updated. In DSRSs, the Reconfigurable Interface (RI) is vital to implement a static routing between a reconfigurable IP and the static part of the DSRS.

3. RELATED WORKS

The main challenges behind the design and implementation of DSRSs lie in three main subjects depicted in the generic architecture presented in Figure 1 and discussed in the following Sections: the Configuration Controller (CC), the Reconfigurable Interface (RI) and the Configuration Port (CP). The quality of the CC and the CP dictates the applicability of a given DSRS. The faster the reconfiguration process, the smaller time penalty is imposed on the implemented application [2]. A careful design of the RI is mandatory to enable reconfiguration to occur at runtime, since transient values on this interface may provoke malfunction of the whole system or even damage the device.

3.1 Configuration Controller

Ullmann [3] proposes a hardware task manager with real-time constraints for DSRSs. This manager is implemented in software and controls the reconfiguration process and context saving. This task manager is also responsible to control the communication among system modules and external devices (sensors and actuators) through a CAN bus.

Resano [4] designed a scheduler using two techniques, named Prefetch-Scheduling Technique and Replacement Technique. The Prefetch-Scheduling Technique receives as input a set of tasks and decides when they will be loaded from memory to be executed. The main goal of this technique is to hide the time penalty to load a reconfigurable IP by analyzing tasks interdependencies and triggering earlier the reconfiguration process. The Replacement Technique is responsible to increase the possibilities to reuse critical tasks in favor of the global system performance.

Mignolet [5] presents an infrastructure for relocatable tasks managing. In this system hardware tasks can have its context interrupted and continue its processing in software, or vice-versa. Given the nature of the target system, a unified communication is needed. The communication among tasks is implemented trough message exchange. The communication between two task can happen in three distinct ways: (*i*) when both tasks are executing in hardware, a NoC is used for communication; (*ii*) when both tasks are executing in software (in the embedded processor), the communication is implemented through an API; (*iii*) when a task is executing in software and another in hardware, a Hardware Abstraction Layer is used to implement the communication.

Griese [6] proposes a run-time reconfiguration manager. This manager was implemented in hardware and it is responsible to control and execute the reconfiguration process in a target platform. Context switching and security mechanisms were also implemented to prevent a possible unsuccessful reconfiguration. Griese's manager stores partial bitstreams in a host computer, which communicates with the target platform through the PCI.

Carvalho [7] proposes a configuration controller named RSCM, completely in hardware. It is composed by the following modules: (*i*) Configuration Memory, stores all partial bitstreams used by the system; (*ii*) Self-Configuration, controls the physical reconfiguration process; (*iii*) Configuration Interface, sends configuration data to the FPGA; (*iv*) Central Configuration Control, receives requests to start the configuration process and provides results in the form of status signals; (*v*) Reconfiguration Monitor, detects situations where reconfigurations need to be

performed; (*vi*) Configuration Scheduler, determines which configuration is the next to be configured.

3.2 Reconfigurable Interface

As stated earlier, a reconfigurable interface is necessary to implement a static routing between a reconfigurable module and the rest of the DSRS. The interface proposed by Palma [8] in 2002 uses two levels of tristate buffers in the input and output pins of the reconfigurable modules. One level of tristates belongs to the reconfigurable module and the other to the communication medium. Due to the scarce number of tristates and the limited number of routings that two tristate buffers can take, it is necessary to manually verify if the modules to be exchanged have the same routing. In order to reduce this manual step, Palma employs a 1-bit data serial bus as communication medium.

Xilinx proposes in [9] a reconfigurable interface called Bus Macro. This macro is composed by 8 tristate buffers and allows the communication of 4 bits between a reconfigurable area and the rest of the system. This macro simplified the routing verification between two tristate buffers of Palma reconfigurable interface.

Huebner [10] also proposes a reconfigurable interface called Bus Macro (different from the Xilinx Bus Macro and herein named Huebner macro), which is a static bus responsible to connect all reconfigurable modules of the system. This reconfigurable interface is composed by two unidirectional busses, each one implemented by a set of CLBs configured as depicted in Figure 2.

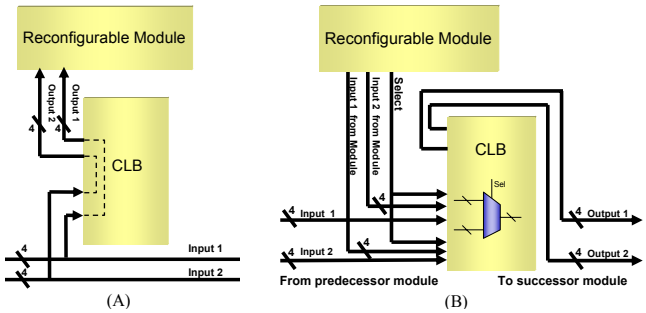


Figure 2 – Huebner's macros: (A) Look-Up-Tables connect through Input-Signal for communication from arbiter to module macro; (B) Look-Up-Tables initialized as Multiplexer for communication from module to arbiter.

3.3 Configuration Ports

The configuration ports used in DSRSs are usually the ones provided by the manufacturers and can be accessed internally or externally from the FPGA. In the case of Xilinx FPGAs, which are used from works [3] to [10], there are three available ports: JTAG, SelectMap and ICAP. JTAG is a serial interface, while the two others can transmit 8-bits data in parallel (Virtex 4 ICAP has a 32-bit configuration port). From these, only the ICAP is an internal port and it can work at 66 MHz and 100 MHz (Virtex 4), while SelectMap and JTAG can operate at 50 MHz and 33 MHz, respectively.

From the presented works, only the work from Ullmann/Huebner explicitly mention to use the fastest communication port, which can be done by a specific hardware connected to the ICAP or through the API implemented by Xilinx and available for MicroBlaze [11] and PowerPC processors.

4. CHOICES AND TRADE-OFFS

This Section discusses design choices available for implementing each element of a DSRS infrastructure, with emphasis on the cost-benefit trade-offs involved in selecting each of these choices.

4.1 Repositories

DSRSs need to have access to repositories able to maintain a potentially large number of partial configurations, in what is often called *configuration memory*. Besides stocking partial configurations, these repositories should offer fast access to its contents for the DSRS such that application requirements be satisfied. 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 BRAM; (*ii*) devices external to the DSRS using static RAM technology, or SRAM; (*iii*) devices external to the DSRS using PROM technology, such as EPROM or Flash devices called generically PROM; (*iv*) devices external to the DSRS using DRAM technology, such as SDRAM and others. Table 1 presents a qualitative comparison among these devices.

Table 1 – comparison of technologies applicable to build DSRS configuration memories.

Memories	Capacity	Speed	Controller Complexity	Cost/Bit
BRAM	+	++++	+	+++
SRAM	++	+++	++	++
PROM	++	+	++++	++
DRAM	++++	++	+++	+

Applications using BRAMs to form a configuration memory need to have a small number of small configurations, due to its limited capacity. Limited applications that benefit from difference-based [9] reconfiguration techniques are among those that can 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 to store more configurations, but 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 the startup of the DSRS. Also, changing the contents of the repository is more complicated than with the other technologies.

4.2 Reconfigurable Interface

Three reconfigurable interfaces were presented in Section 3.2. The main drawback of Palma´s work [8] is to manually route the interface between the reconfigurable area and the fixed area. The Xilinx Bus Macro [9] reduces manual routing, however, it also uses tristate buffers, which are scarce resources in Xilinx FPGAs.

The use of such resources constrains the system design with complex reconfigurable interfaces.

Huebner [10] points that the routing tool sometimes does not respect the constraints set by the designer and crosses the boundary between the reconfigurable module and the fixed part of the system. Due to this problem Huebner developed a static bus implemented with LUTs. One advantage of this approach is to insulate computation from communication, thus preventing any system interruption. However, this reconfigurable interface imposes the use of a bus-based communication medium.

The present work proposes a new reconfigurable interface, based in LUTs as Huebner's macros, but with the difference to not impose the use of a specific communication medium. The reconfigurable interface is composed by two unidirectional macros, depicted in Figure 3. The first one, named F2R, is responsible to send data from the fixed part of the system to a reconfigurable module, while the second one, named R2F, implements the inverse direction of communication. Both macros allow the parallel transmission of 8 bits. The F2R macro is configured with the identity function, while the R2F uses a special logic to prohibit transient switchings during the reconfiguration process from reconfigurable to fixed areas.

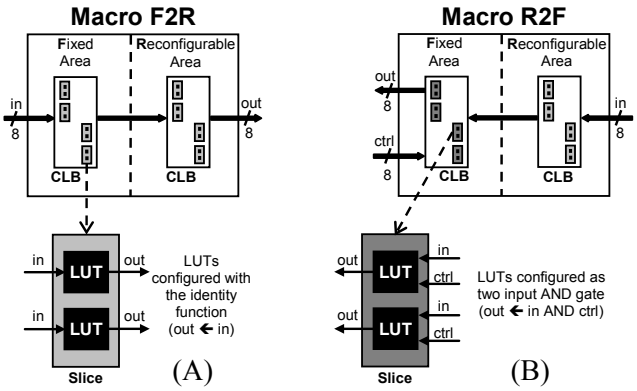


Figure 3 – Macros proposed in current work: (A) Fixed to Reconfigurable; (B) Reconfigurable to Fixed.

4.3 Configuration Ports

As already stated in Section 3.3 the most popular devices used as basis for implementing DSRSs are Xilinx FPGAs, specially the Virtex and Spartan families. In these devices, 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 (by using external wiring connecting some of the reconfigurable device pins to them) most of these devices have available an Internal Configuration Access Port, or ICAP. The ICAP usually constitutes the best choice for supporting the construction of DSRS, once it can be accessed from inside the reconfigurable device by user logic.

4.4 Configuration Controller

DSRSs require an internal module to manage the internal reconfiguration process, the configuration controller (CC). Section 3.1 already discussed some propositions of CCs available in the literature. One of these is the RSCM [7]. Two versions of this CC have been built by the Authors: (*i*) a version fully built in hardware; (*ii*) a version developed mostly in software. Table 2 compares these two implementations qualitatively.

Table 2 – Qualitative comparison of the RSCM configuration controller implementations.

Characteristic	RSCM (hardware)	RSCM-S (software)
Configuration Speed	Milliseconds	Milliseconds
Area	Requires additional hardware	If processor available, small area overhead (ICAP and macro controllers required)
Easiness of modification	Complex / extra area	Simple / software modification

RSCM-S is in average three times slower than the RSCM hardware. This disadvantage is related to the inefficiency of the current API furnished by Xilinx to give access to ICAP. This API requires the CC 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 RSCM sends data directly from an external memory to ICAP, leading to smaller reconfiguration time.

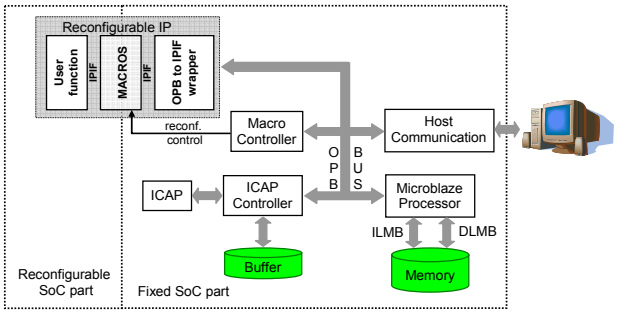
The RSCM-S runs on an embedded 32-bit RISC designed by Xilinx, MicroBlaze. The structure of RSCM-S includes, besides the processor, peripheral device controllers, memory and a communication medium. Obviously, if this infrastructure is applied to the configuration control alone, the approach could hardly 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 configuration controller tasks, the additional hardware in fact requires an area smaller than that needed by RSCM. Given the assumptions above and if the application reconfiguration time requirements are not too stringent, RSCM-S can be usefully applied.

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 configuration controller, a software implementation is definitely more flexible. Complex tasks can be easily accommodated through programming. Examples of such functionalities are bitstream compression and on-the-fly decompression, configuration scheduling policies, and support to configuration preemption. A hardware-only implementation as RSCM, would require restructuring the CC design, realizing the CC re-synthesis and would probably increase the area overhead of the controller.

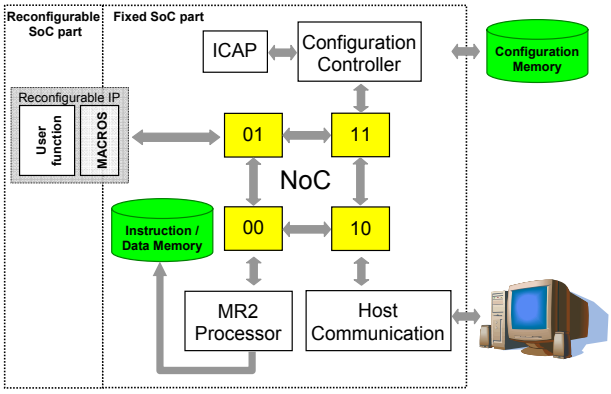
5. DSRSs IMPLEMENTATIONS

This Section presents the implementation of two DSRSs case studies. The first case study, depicted in Figure 4(A), employs: (*i*) configuration controller implemented in software, RSCM-S; (*ii*) partial bitstreams stored in internal BRAMs; (*iii*) ICAP configuration port; (*iv*) Xilinx API to access ICAP; (*v*) OPB bus as communication infrastructure. In contrast, the second case study, illustrated in Figure 4(B) employs: (*i*) configuration controller implemented in hardware, RSCM; (*ii*) partial bitstreams stored in external SRAMs; (*iii*) ICAP configuration port; (*iv*)

direct access to the ICAP port through a dedicated hardware controller; (*v*) Hermes NoC [12] as communication infrastructure.



(A) bus-based architecture



(B) NoC-based architecture

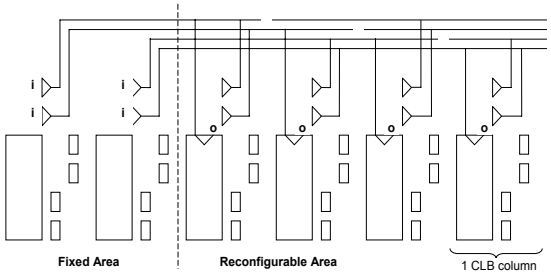
Figure 4 –DSRSs architectures.

Table 3 compares the area required to implement different memory controllers and the typical capacity of such memories, used to store partial bitstreams. Internal BRAM and external SRAM are both static memories, requiring insignificant area to implement their controllers. SDRAM require more complex controllers, offering to users larger memory capacity. Typical SRAM capacity, 1MB, is sufficient to store approximately ten partial bitstreams in a XC2VP30 device. SDRAMs were not used in the case studies as configuration repository due to the area to implement this controller.

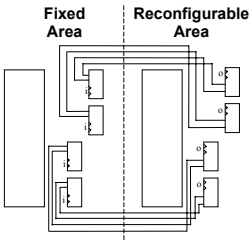
Table 3 – Area required to implement different memory controllers and the typical capacity of each one.

Memory Type	Slices (Area)	Typical Capacity
BRAM	4	306KB (XC2VP30)
SRAM	8	1MB
SDRAM	373	64MB

In both experiments LUT-macros were used in the reconfigurable interface. Huebner macros [10] are in fact a bus, restricting the architecture. Macros developed by Xilinx [9] uses a larger area when compared to the LUT-macros proposed in current work. Figure 5 illustrates the difference: the Bus Macro consumes 6 CLB columns, being two in the fixed area and four in the reconfigurable area; while the LUT-macro occupies 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: Bus Macro is 4-bit wide and LUT-macro 8-bit wide. The advantages of LUT-macro justify its usage in both DSRSs case studies.



(A)



(B)

Figure 5 – Physical implementation of macros configured to communicate data from left to right: (A) Bus Macro [9]; (B) F2R Macro proposed in current work.

Another common choice in both experiments is the configuration interface: ICAP port. The first case study uses the Xilinx API to access the ICAP port, while the second case study uses a dedicated module that was developed to access the ICAP port. The Xilinx API is a slow process for two reasons: (*i*) a buffer is inserted between the repository and the reconfiguration port; (*ii*) the MicroBlaze processor controls all steps. The dedicated hardware module implemented in the second case study acts as a DMA controller. The processor indicates the address of the partial bitstream in memory to the controller, which reads the words from memory transferring them directly to the ICAP port.

It is used a partial bitstream with 5 CLB columns in a XC2VP30 device to compare the performance to reconfigure a partial bitstream with both methods. The reconfiguration using the Xilinx API consumed 34.76 *ms* and the dedicated hardware module 9.98 *ms*. Despite the fact the hardware module is in this implementation is 3.5 faster, the software implementation has several advantages, such as flexibility to implement/modify features (e.g. scheduling, bitstream compression and context switching) by only writing a new piece of code, without hardware synthesis. The present disadvantage, high reconfiguration time, is due to the Xilinx API responsible to access the ICAP module. Write a new API to access the ICAP module is a feasible work, which can reduce dramatically the reconfiguration time.

Table 4 compares the area to implement the configuration controller in experiment 1 (with MicroBlaze) and experiment 2 (with dedicated hardware). If a processor is already available in the system (as MicroBlaze or PowerPC), the area of the software configuration controller represents the area of the ICAP Controller and Macro Controller, resulting in 1.82% of FPGA area usage, against 3.44% from configuration controller implemented in hardware.

Table 4 – FPGA area report for two configuration controllers implementations: (A) software; (B) hardware.

(A) Module	Number of Slices	Number of FFs	Percentage of Slices
MicroBlaze	571	366	4.17%
MicroBlaze support	160	75	1.17%
ICAP Controller	151	155	1.1%
Macro Controller	99	136	0.72%
Total	**981**	**732**	**7.16%**

(B) Module	Number of Slices	Number of FFs	Percentage of Slices
MR	11	20	0.08%
EC	42	69	0.31%
CCC	71	97	0.52%
AC	130	236	0.95%
MC	216	329	1.58%
Total	**470**	**751**	**3.44%**

6. CONCLUSIONS

Architectures to implement reconfigurable systems have been largely discussed in the literature recently. It is possible to enumerate different ways to implement such systems, but one common feature pervades in most works: the use of Xilinx components. This work has discussed trade-offs and choices to implement dynamic self-reconfigurable systems. Table 5 summarizes the proposed choices.

Table 5 – Recommended infrastructure choices for DSRSs.

Infrastructure Element	Recommended Choice
Configuration Controller	Software + new API (future work)
Reconfigurable Interface	LUT-Macro
Repository	External SRAM
Reconfigurable Port	ICAP
Interconnection Infrastructure	NoC (Scalability)

The experiments show that implementing the *configuration controller in software* gives to the user more flexibility to implement new algorithms (e.g. scheduling), allows to modify and insert new functionalities (e.g. bitstream compression, context switching) only writing a new piece of code, without hardware synthesis. The present disadvantage, higher reconfiguration time, is due to the Xilinx API responsible to access the ICAP module. Write a new API to access the ICAP module is a feasible work, which can reduce the reconfiguration time.

The *reconfigurable interfaces should use LUT-macros*, instead the Xilinx Bus Macro, since they allow to implement wider interfaces between modules, occupies less area, and can be easily adapted to support insulation during the reconfiguration process.

It is recommended to use *external static RAM to store the partial bitstreams*, 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 bitstreams. It is not recommended to waste internal FPGA memory with partial bitstreams, since the capacity of such memories is too small.

ICAP is the only available choice to implement the reconfiguration port without employing external circuitry. Ports as JTAG and SelectMAP may be used, but they require external components.

Few works employ network-on-chip infrastructure as communication infrastructure. *NoCs are the recommended option*, mainly due to the scalability, essential feature in future SoCs.

These choices define a reference architecture for reconfigurable architectures research. Short-term work includes: improve the configuration controller implemented in software, with a faster access to the ICAP and include bitstream decompression to reduce the memory requirements. Future work includes evaluate the control protocols necessary to include reconfigurable IPs in a NoC, and link hardware reconfiguration to operating systems (OS), enabling OS to schedule both hardware and software tasks.

7. ACKNOWLEDGMENTS

This research was supported partially by CNPq (Brazilian Research Agency), project 307655/2003-2.

8. REFERENCES

[1] Keutzer, K.; et al. “System-Level Design: Orthogonalization of Concerns and Platform-Based Design”. IEEE Transactions on CAD of Integrated Circuits and Systems, v. 19(12), pp. 1523-1543, 2000.

[2] Wirthlin, M.; Hutchings, B. “Improving Functional Density Through Run-Time Constant Propagation”. In: FPGA’97, pp. 86-92.

[3] Ullmann, M.; et al. “An FPGA Run-Time System for Dynamical On-Demand Reconfiguration”. In: IPDPS’04, pp. 135-142.

[4] Resano, J.; et al. “Specific Scheduling Support to Minimize the Reconfiguration Overhead of Dynamically Reconfigurable Hardware”. In: DAC’04, pp. 119-121.

[5] Mignolet, J.; et al. “Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-On-Chip”. In: DATE’03, pp. 986-991.

[6] Griese, B.; et al. “Hardware Support for Dynamic reconfiguration in Reconfigurable SoC Architectures”. In: FPL’04, pp. 842-846.

[7] Carvalho, E.; Calazans, N.; Moraes, F.; Mesquita, D. “Reconfiguration Control for Dynamically Reconfigurable Systems”. In: DCIS’04, pp 405-410.

[8] Palma, J.; Mello, A.; Möller, L.; Moraes, F.; Calazans, N. “Core Communication Interface for FPGAs”. In: SBCCI’02, pp. 183-188.

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

[10] Huebner, M.; Becker, T.; Becker, J. “Real-Time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration”. In: SBCCI’04, pp. 28-32.

[11] Xilinx, Inc. “MicroBlaze Processor Reference Guide”. Reference Guide v4.0 edition, Aug. 2004, 132 pages.

[12] Moraes, F. et al. “Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip”. Integration, the VLSI Journal, v.38(1), 2004, pp. 69-93.