The Shining embedded system design methodology based on self dynamic reconfigurable architectures

C. A. Curino, L. Fossati, V. Rana, F. Redaelli, M. D. Santambrogio, D. Sciuto
Dipartimento di Elettronica e Informazione, Politecnico di Milano (Milan, IT)
{curino,fossati,rana,santambr,sciuto}@elet.polimi.it, redaelli@dresd.org

ABSTRACT

Complex design, targeting System-on-Chip based on reconfigurable architectures, still lacks a generalized methodology allowing both the automatic derivation of a complete system solution able to fit into the final device, and mixed hardware-software solutions, exploiting partial reconfiguration capabilities. The Shining methodology organizes the input specification of a complex System-on-Chip design into three different components: hardware, reconfigurable hardware and software, each handled by dedicated sub-flows. A communication model guarantees reliable and seamless interfacing of the various components. The developed system, stand-alone or OS-based, is architecture-independent. The Shining flow reduces the time for system development, easing the design of complex hardware/software reconfigurable applications.

1. INTRODUCTION

Wayne Wolf in [1] states that the primary task of HW/SW codesign is to increase the predictability of embedded system design, providing both analysis and synthesis methods. This requires new methodologies to support the designer in the definition of architectures which typically foresee a general-purpose microprocessor to be used in conjunction with Intellectual Property Cores (IP cores), typically implemented as custom hardware components. Functionalities implemented in hardware, generally, yield better performance but with higher costs in terms of chip area usage, while those that run as software have, in general, worse performance but do not require additional hardware resources. Moreover, reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), introduce another degree of freedom in the design workflow: the designer can exploit the fact that the system can now autonomously and dynamically modify the functionality performed by the IP-Core. We will refer to dynamically reconfigurable IP-Cores as RPEs (Reconfigurable Processing Elements). The fast introduction of new standards and protocols, in particular in the multimedia field, is another relevant driver towards reconfigurable solutions. In a dynamically reconfiguration scenario, the Reconfigurable Processing Elements (RPEs) functionality identification, its generation and run-time hot-plug mechanisms are issues of primary concern. Another relevant issue during the development of the system is to enable fast validation of new RPEs in the actual system. The proposed methodology speeds-up the RPE generation by reducing the impact of interfacing and plugging of these elements into the system. In this way the attention of the designer is focused entirely on the development of the desired functionality implementation with almost no burden for communication infrastructure issues. The methodology proposed is an extension of a reconfigurable design flow [2]. This guarantees the new methodology to be actually exploited on FPGA-based embedded systems to exploit both the potentiality of a mixed HW-SW and reconfigurable HW solution. The overall tool-chain has been tested on the Xilinx VirtexIIPro FPGAs: VIIP7 and VIIP20. The paper is organized as follows: Section 3 presents the proposed methodology, the experimental results are described in Section 4 while Section 5 draws the conclusions and briefly discuss future developments.

2. RELATED WORKS

The *PipeRench* architecture, proposed in [3], introduces the concept of hardware virtualization to provide the possibility of implementing a design of any size on a given configurable pipeline architecture of arbitrary capacity. The *PipeRench* system provides both the extremely high-speed reconfiguration necessary for hardware virtualization and the compilation tools for this architecture. However the *reconfigurable pipeline* structure introduces some relevant constraints that limit the freedom of the design and it is not *portable* to generic architectures. In [4], [5] the hardware subsystem of the reconfiguration control infrastructure sits on the on–chip peripheral bus (OPB). The microprocessor, PowerPC or Microblaze, communicates with this peripheral over the OPB bus. The hardware peripheral is designed to provide a *lightweight* solution to reconfiguration. It employs a read/modify/write strategy where only one frame of data is used at one time. In this way external memory is not needed to store a complete copy of the configuration memory. The program installed on the processor requests a specific frame, then the control logic of the peripheral uses the ICAP, the internal configuration access port, to do a read-back and loads the configuration data into a dual–port block RAM. One block RAM can hold an XC2V8000 data frame easily. When the read–back is complete, the processor program directly modifies the configuration data stored in the BRAM. Finally, the ICAP is used to write the modified configuration data back to the device. The software subsystem is implemented using a layered approach. There are functions for downloading partial bitstreams stored in the external memory, for copying regions of configuration memory, and parsing it to a new location [4]. This solution is very interesting but it does not support external memory to store the bitstreams therefore it can be used just to support small configurations. In [6] the authors proposed an architecture (based on a 1D self reconfiguration approach) which is logically divided in two parts: a static side and a reconfigurable one. The static side contains a standard

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

IBM CoreConnect technology [7] and two different buses: the OnChip Peripheral Bus (OPB) and the Processor Local Bus (PLB). The communication between the static and the reconfigurable sides is obtained by exporting from the static side the necessary OPB signals used to define the correct binding of each reconfigurable module to it. The Caronte architecture has two main weaknesses in the communication and the portability over different FPGA Families. The communication infrastructure between the static side and the reconfigurable one implies that, while a reconfiguration process takes place, inter-modules communications are disrupted. The proposed infrastructure also limits the number of reconfigurable modules. The second problem is caused by the instantiation of a Power-PC processor inside the static side. This limits the portability of the architecture to only FPGAs containing this processor inside their die.

3. THE SHINING METHODOLOGY

The proposed methodology aims at defining a specification-to-bistream design flow based, as far as possible, on standard tools. Throughout this paper we will refer, where needed, to a reference reconfigurable architecture, internally designed and developed. However the methodology itself is fully general and it has been successfully applied to different architectural and SoC solutions such as Raptor2000 [8], Caronte [6] and Lightweight [5]. The reference architecture is based on the 1D reconfiguration approach used into the VirtexII and VirtexIIPro Xilinx FPGA families. The architecture foresees a static and a reconfigurable portion; the static one is typically designed using standard tools, such as EDK [9], or manually fully described in VHDL. It is composed of a processor (that can be hardcore, i.e., a PowerPC, or softcore, i.e.,a Xilinx MicroBLAZE, from now one we will refer to the PowerPC solution, just for simplicity) and a set of cores used to implement an appropriate bridge to the reconfigurable portion. The IBM CoreConnect [7] has been chosen as main communication infrastructure. The processor will be used to execute the software side of the application and to correctly manage the necessary calls to the hardware components (with the corresponding reconfiguration, if necessary) designed to speedup the overall computation. The reconfigurable part of the proposed architecture is a portion of the FPGA area on which different functionalities can be mapped, with only the requirement to implement an appropriate bus interface. The reference reconfigurable architecture is physically implemented in three different layers. The first layer contains the communication infrastructure layer between the static and the reconfigurable side, while the second one contains CLBs and Switch Matrices and consequently all user logic, but also the CoreConnect components are implemented at this level. Finally, the last layer is the clock level that is routed at a different level with respect to the other signals [10, 11]. Such an architecture is perfectly suited to support the two main issues of our methodology: mixed HW-SW execution and dynamic reconfigurable system design. Figure 1 shows the flow of the proposed methodology. Different solutions can be obtained by tuning the parameters (i.e. the communication infrastructure) used to define the proposed methodology. The flow generates different feasible implementations for each required functionality, HW or SW, with different bus solutions. These solutions will be defined using the proposed architecture, implementing a dynamic self reconfigurable architecture, deeply described in

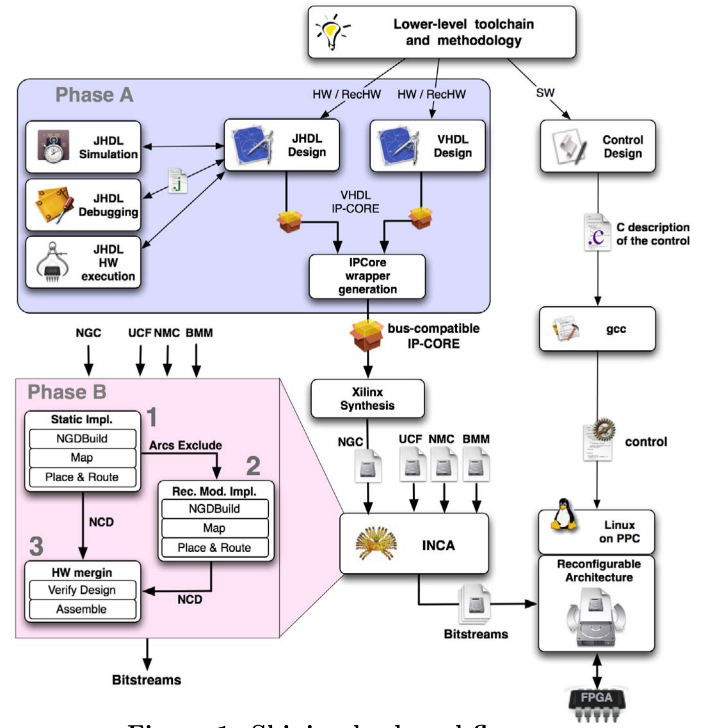

Figure 1: Shining back-end flow

Section 3.2 (Phase B in Figure 1) or a *standard* embedded system, characterized by a general purpose processor and a set of HW cores, described in 3.1, Phase A in Figure 1. The SW side of the desired application, extended with the reconfiguration manager, as described in Section 3.3, can be implemented either as stand-alone software application or to run on an Operating System (OS) that provides reconfiguration mechanisms. On one hand, the first choice is oriented toward the creation of a specific solution that is deeply optimized for a specific problem, even if it requires a large investment in terms of design and implementation efforts, and considerably increases the time to market, since it is necessary to rewrite the whole application each time the context changes. The second choice, on the other hand, can be followed to increase both the level of abstraction and the flexibility of the whole system, since it exploits the classical services that an OS can provide, such as processes scheduling techniques or inter-process communication systems, applying them to improve the reconfiguration management.

3.1 The IP-Core generation phase

Aim of this phase is to build a complete IP-Core from its core logic. This task is automatically performed through three steps by a tool named IPGen: *registers mapping*, *address spaces assignment* and *signals interfacing*. The Registers mapping step is needed because each core may have different (number, type, size) set of signals. In this scenario the most suitable solution is the use of a standard set of signals for the communication with the rest of the system. To avoid undesired interferences with the core logic during the set-up, to map standard to core-specific signals, each signal must be stored temporally. This decoupling is done by means of a set of registers interposed between the core signals and the standard system one. *The second step* that has to be performed is the address space assignment where

each register is now assigned to a specific address. *The last step* consists in the signals interfacing phase. Target bus signals are mapped to registers. After the execution of this sequence of steps, the IP-Core is ready to be bound to the target bus and has a proper interface. The set of results presented in table 1 concerns several types of components, starting from some small IP-Cores such as an IrDA interface to more complex examples, e.g a Siemens Mobile Communication description of a complex ALU, a video editing core that changes the image coloring plane from RGB to YCbCr. Table 1 presents some relevant results, considering both the input core (the first row for each core), that represents the core logic, and the obtained component (the second row), that is the final IP-Core produced by the *IPGen* tool. For each one of them, the size in terms of 4-INPUT LUTs and the number of occupied slices are illustrated, both as absolute values and as the percentage with respect to the total size of the FPGA. Columns labeled with ar_i and d_i, present the number of available resources (ar_i) given the minimum area constraints to implement the core as reconfigurable elements, and the density (d_i) computed as the ratio between the number of slices used to implement the core and the number of available resources, ar_i. The last column shows the time needed by *IPGen* to create the IP-Core. On one hand the relative overhead due to the interface of the core logic with the bus infrastructure is acceptable, both for the 4-INPUT LUTs and for the occupied slices, especially when the core size is relevant. This allows the use of the generated IP-Cores in the final reconfigurable system without wasting too much space on the reconfigurable devices. In order to create the IP-Core, starting from input Core, IPGen needs an execution time which is almost constant and on average of 0.065 seconds.

Table 1: IPGen benchmarks

IP-Core	# Columns	Ratio	Slices	Ratio	ar_i	d_i	Time (s)
IrDA	1		11		136	0.081	
	1	9.73	103	9.36	136	0.758	0.045
FIR	2		153		272	0.562	
	2	1.13	173	1.13	272	0.636	0.058
RGB2YCbCr	7		913		952	0.959	
	7	1.21	940	1.03	952	0.987	0.063
Complex ALU	7		950		952	0.997	
	8	1.19	1079	1.14	1088	0.991	0.071

3.2 INCA Design Flow

Phase A, in Figure 1, produces as output all the necessary files used to describe the desired system, both in its static and reconfigurable side. This section presents the *Physical Implementation* phase, named INCA (Phase B in Figure 1). Between the highest part of the flow, used to generate the VHDL descriptions, and the INCA one (the physical implementation), there is a classical synthesis phase (implemented using standard synthesis tool i.e., XST) used to create all the necessary inputs (i.e., macro hardware, netlists, constraints file) for the INCA flow from the generated input files. INCA can be used to support both the Xilinx Early Access Partial Reconfiguration-based, [12], and the Module-based, [11], reconfigurable architecture design flow. This phase has been organized into three different stages: static part implementation, reconfigurable modules implementation and hardware merging. The former one accepts as input the input description files generated as described in Sections 3.1. Aim of this stage is the physical implementation of the static side of the final architecture. A second output, working with the EAPR flow, is represented by the information, *arcs exclude*, of the static side components that are placed into the reconfigurable modules region i.e., routing, CLBs usage. The **reconfigurable modules implementation** stage needs for each module the corresponding description files and the *arcs exclude* (produced in the previous stage) information. This stage defines the physical implementation of each reconfigurable component that has to be passed as input to the hardware merging phase. It needs also to be repeated a number of times, equal to the number of reconfigurable components, to define the final architecture. Finally, the **hardware merging** stage produces as result the merging of the outputs produced by the two previous stages. Based on these considerations we can say that the adopted solution consists in the generation of both a complete bitstream, based on the top that configures the system and a set of empty modules. Then for each module two partial bitstreams have to be created: one is used to configure it over an empty module and another one to restore the empty configuration. In such a situation we do not need, at compile time, to compute and to know all the possible reconfiguration combinations between all the modules we have and we can also support modules relocation [13][1] saving memory to store useless and redundant reconfiguration files. Table 2 reports the average execution time for each phase composing the INCA flow. We report the results for all the devices used in our tests: Xilinx Spartan3, Virtex II Pro and Virtex 4.

Table 2: INCA execution times

Phase B	S200		VP7		FX12	
(#)	(s)	%	(s)	%	(s)	%
1	133.86	18.8%	224.91	18%	1060.28	39.0%
2	110.72	14.1%	165.69	13.3%	249.72	9.1%
3	466.19	65.5%	852.13	68.5%	1407.36	51.8%
Total	710.77	100.00%	1242.73	100.00%	2717.36	100.00%

3.3 The Software Architecture

This phase has been introduced to *merge* the software, that has to be executed on the processor to manage the reconfiguration. Output of this last phase is the complete start-up configuration bitstream and all the information that has to be provided as input to the Bitstream Creation Phase to compute all the partial reconfiguration bitstreams necessary to implement the desired self reconfigurable system. The actual version of the reference reconfigurable architecture is based on GNU/Linux operating system, which is a complete multitasking operating system. The operating system considers the reconfiguration process as an autonomous thread of computation. For this reason, the software reconfiguration support and the functions which deal and manage the hardware are separated. In this case, the application code runs as a user process in the system; this means that it does not have direct low level access to the hardware, but it has to pass all the requests through operating system calls (read, write, etc...). Therefore, as far as reconfiguration in concerned, the OS itself must take care of the communication with the ICAP, by exporting an interface to user processes. The basic idea is to hide the reconfiguration process also to the application. Parts of the initial specification have been moved into RPEs therefore the corresponding code has been substituted with a set of OS calls used to interface the code,

[1] we are not going to describe in detail this scenario, since it is out of the scope of this paper

executed on the processor, with the hardware. Whenever a request for an RPE which has not been yet configured, the OS will take care of the reconfiguration process of the correct RPE and it will correctly manage the new hardware as soon as it will be available. Since the GNU/Linux kernel does not have any kind of support for ICAP, we developed a driver for the ICAP peripheral. Linux allows userspace programs to access devices via special files, located under the `/dev` directory. Each device is assigned a couple of numbers as id, indicating the driver managing the device (the *major* number) and the id of the specific device (the *minor* number); further, they are also divided in *character* and *block* devices, based on the kind of access they support. When a kernel driver registers a major number, all access requests to the corresponding devices are directed to it, and hence it must implement handlers for various system calls: open, close, read, write, and so on. The ICAP module, on startup, registers a character device major number (by default 120) and reserves the memory–mapped address space corresponding to the ICAP device; the base address can be specified as a parameter when loading the module. At this point it is possible to create a device file with major number 120 and minor 0, for example `/dev/icap`, that processes can access to execute reconfiguration.

4. EXPERIMENTAL RESULTS

To validate the discussed flow we applied it to several real-life applications. The result has been successful and we have been able to automatically derive a collection of working bitstreams, each describing a different implementation of the original application. The bitstreams have been downloaded onto a Xilinx FPGA, a Virtex II Pro (xc2vp7). The clock frequency of the board has been set to the default value of 100MHz. Although the PLB and the OPB can run at different frequencies, the results shown in Tables 3, 4 and 5, have been obtained by using the same frequency for all the buses. This to enables a performance comparison not being affected by different running frequencies but depending on the bus type only. Table 3 shows a solution in which the

Table 3: Canny algorithm

Modules	Bus	N° of cycles	Speed Up
Software		18170678	1
4 pixels at a time	32 bits OPB	18757069	0,97
Pipelined Core	32 bits OPB	9124777	1,99
Pipelined Core	32 bits PLB	8729977	2,08
Pipelined Core	64 bits PLB	7532874	2,41

processor sends one pixel at a time to the IP-Core obtaining better performance with respect to the one using a 4-pixel-wide communication. The problem of the 4-pixels solution is the time spent by the application in data preparation: a task composed of few, but expensive, operations. These two variants of the same application use non-pipelined IP-Cores, as a consequence the performance improvement (if any) is minimum; the last three variants presented, on the contrary, exploit pipelined cores. Here the processor operates sending rows of pixels in a pipelined fashion. The overall result is a reduction of the communication overhead of about 85%. The results are really positive also on the OPB bus. The improvement that can be obtained by exploiting the PLB bus is a minimum part the total time, since it can be decreased only of the time used through both the OPB bus and the bridge. All the presented variants share a fixed bus width of 32 bits. This, since the pixels matrix is 56 bits per row (7 by 8 bits per pixel), requires the processor to invest 2 communication cycles for each row. Such a problem can be solved by increasing the bus width to 64 bits; the performance improvement is shown in the last line of Table 3. The limitations of non-pipelined solutions are evident also in

Table 4: Sobel Convolution

Modules	Bus	N° of cycle	Speed Up
Software		2576102	1
4 pixels per time	32 bits OPB	3268152	0,79
Pipelined Core	32 bits OPB	1492154	1,73
Pipelined Core	32 bits PLB	1476992	1,74
Pipelined Core	64 bits PLB	1531765	1,68

some test on Sobel convolution shown in Table 4. Furthermore, the Sobel convolution is very simple and the communication overhead has a heavy impact. The three variants using pipelined IP-Cores, coherently with the results of the test on the Canny algorithm, have a bigger performance improvement. The PLB-based solution again performs just a little better than the OPB one. However, since Sobel convolution is based on 3x3 pixels matrix, a 32 bits bus solution is enough, and the overhead of 64 bits variables impacts negatively on the performance. Finally, Table 5 presents the

Table 5: Laplace Convolution

Modules	Bus	N° of cycle	Speed Up
Software		3900864	1
4 pixels per time	32 bits OPB	6037218	0,65
Pipelined Core	32 bits OPB	2923865	1,33
Pipelined Core	32 bits PLB	2766401	1,41
Pipelined Core	64 bits PLB	2055701	1,90

results for the Laplace convolution: the same considerations made for the Sobel convolution apply, with the only exception of the bus width. Laplace convolution is based on a 5x5 pixel matrix and so a single row occupies 40 bits, as a result a 64 bits bus width leads to one-cycle communications and so better performance.

4.1 The Reconfigurable Solution

Aim of this section is to show that the proposed flow is platform-independent and that it can be used to design different reconfigurable architectures (both internal and external reconfigurations are supported). Again we will show that the flow is processor-independent (PowerPC or Microblaze) and that it can use different placement constraints (1D and 2D) and different reconfigurable constraints. As a target device for the first two examples we consider the Digilent Starter Board, the reprogrammable device is a S3 FPGA. The S3 does not contain a hard-processor, as a consequence a Microblaze soft-processor has been used. The second and third examples target the Avnet Evaluation Board, which contains a Virtex-II Pro V2P7 FPGA. Finally, the target device of the last two examples is again a Avnet Evaluation Board, where the PPC hard-processor of the V2P7 FPGA substitute the Microblaze soft-processor. For each one of these settings two tests have been run featuring respectively: two modules directly connected with external components or two modules communicating through the bus to the static portion of the architecture. Furthermore, for each setting the size of reconfigurable and static part of the architecture has been incrementally decreased, to test different shapes for each of them. The presented approach has been applied to automatically develop several reconfigurable architectures.

Table 6: Experimental results

FPGA	S3	S3	Virtex-II Pro	Virtex-II Pro	Virtex-II Pro	Virtex-II Pro	Virtex 4
Processor kind	Soft-processor	Soft-processor	Soft-processor	Soft-processor	Hard-processor	Hard-processor	Soft-processor
Processor name	Microblaze	Microblaze	Microblaze	Microblaze	PowerPC 405	PowerPC 405	Microblaze
Static part region	Without constraints	From X8Y0 to X39Y47	From X24Y0 to X63Y79	From X24Y0 to X63Y79	From X24Y0 to X63Y79	From X24Y0 to X63Y79	From X0Y10 to X23Y99
Size of the static part	1514 of 1920 slices (78 %)	1464 of 1920 slices (76 %)	1698 of 4928 slices (34%)	2837 of 4928 slices (34%)	2837 of 4928 slices (58%)	2837 of 4928 slices (58%)	1712 of 5472 slices (32%)
Fragmentation index	1.5 %	4.7 %	46.9 %	46.9 %	11.3 %	11.3 %	21 %
Complete bitstream size	127 KB	127 KB	548 KB	548 KB	548 KB	548 KB	582 KB
First reconfigurable region	From X4Y0 to X7Y47	From X0Y14 to X3Y47	From X0Y0 to X11Y35	From X0Y12 to X11Y41	From X0Y0 to X11Y35	From X0Y12 to X11Y41	From X28Y0 to X43Y39
Maximum module size in the first reconfigurable region	116 slices	116 slices	116 slices	116 slices	116 slices	116 slices	640 slices
Fragmentation index	39.6 %	14.7 %	73.2 %	67.8 %	73.2 %	67.8 %	48%
Partial bitstream size	8.51 KB	8.51 KB	36 KB	36 KB	36 KB	36 KB	72 KB
Second reconfigurable region	From X12Y0 to X15Y47	From X4Y14 to X7Y47	From X12Y44 to X23Y79	From X12Y44 to X23Y73	From X12Y44 to X23Y79	From X12Y44 to X23Y73	From X28Y64 to X43Y103
Maximum module size in the second reconfigurable region	115 slices	115 slices	115 slices	115 slices	115 slices	115 slices	640 slices
Fragmentation index	40 %	15.5 %	73.4 %	68.1 %	73.4 %	68.1 %	48%
Partial bitstream size	8.51 KB	8.51 KB	36 KB	36 KB	36 KB	36 KB	72 KB

All the solutions have been successfully tested on the target devices. Table 6 shows, grouped in 4 sections, the most relevant experimental results. The **first section** presents general information such as the name of the FPGA on the target board, the kind of processor included in the static part of the architecture (a soft-processor or a hard-processor) and the name of the processor. The **second section** concerns the static part of the architecture, and it indicates the location of the static region, the fragmentation index of the static part (that is the percentage of space that is not really used by the static part) and the size of the bitstream that is necessary to initially configure the FPGA with the static part region and two reconfigurable modules. The **third section** corresponds to the first reconfigurable module and it presents the location of the first reconfigurable region, the maximum size of modules that will be plugged in the first reconfigurable region, the fragmentation index of the first reconfigurable region part, the maximum size of the bitstream that is necessary to reconfigure a module of the first reconfigurable region on the FPGA. The **fourth section** concerns the second reconfigurable module and it presents the same information of the previous section referred to the second reconfigurable module. All the locations are expressed with the notation *From XaYb to XcYd*, where *(a,b)* is the coordinate of the low-left corner of the region, while *(c,d)* represents the high-right hand corner of the region. Furthermore, Figure 2 show the FPGA layout of the reconfigurable architecture. A VirtexIIPro reconfigurable architecture is shown in Figure 2. This architecture has been designed using a 2D placement assignment and a 1D embedded reconfigurable technique. The 1D approach is mandatory since the architecture has been implemented using a VIIP, that does not support 2D reconfigurations. The VirtexIIPro-based architecture can be used to define a self partial dynamic reconfigurable architecture for all those applications that can benefit from it, since it is provided with a ICAP port. Several cores, spanning from simple functional units to more complex ones such as RGB converter, FIR (the last two have been used as part of a complete edge detector system), have been implemented as reconfigurable components. The evaluation framework has been defined using an architecture composed of two reconfigurable cores reconfigured using a self partial dynamic reconfiguration technique managed by the Microblaze. In the generic evaluation scenario composed of 5 (this value can be

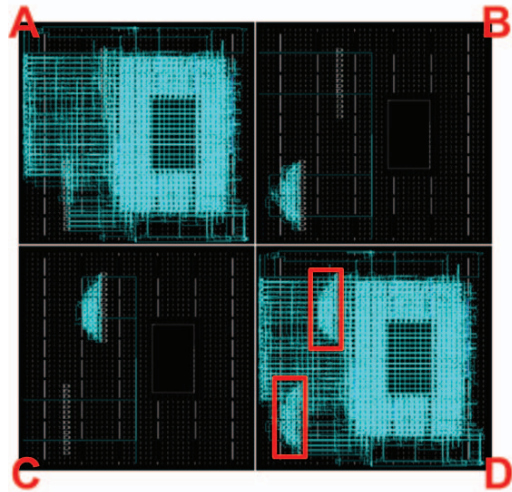

Figure 2: VIIP-based Reconfigurable Architecture

modified without loosing in generality) functionalities that have to be mapped in a known sequence, the start-up configuration is characterized by the presence of a functionality in each module, while the reconfiguration bistreams have to reconfigure just one of the modules leaving the others unaltered. The experimental results have been performed on the Virtex-II Pro Development Board using the architectural solution proposed in Figure 2, with the processor running at 66 MHz and the average measured throughput is of 1.496 MByte/sec.

4.2 A simple complete application

Next, we demonstrate a complete example from the Digital Image Processing area. Several applications in this domain are characterized by data intensive kernels that involve a large number of repetitive operations on the input images. This lead us to consider an implementation where all compute intensive tasks are mapped onto the reconfigurable hardware. The application chosen to validate the overall solution is the *edge detection* problem, computed on sequential frames, e.g. for a *motion detection* application [14]. The edge detector we used in our experiments is the *canny edge detector* which is composed of four main steps: image smoothing (f_a), gradient computation (f_b), non-maximum suppression (f_c) and finally the hysteresis threshold (f_d).

We decided to adapt our execution model to be able to justify the reconfiguration approach using a model similar to the one proposed in [15]. The idea is to iterate the execution of each module a certain number of times, and in such a way to obtain modules whose running time is comparable to the reconfiguration time of other modules, thus hiding reconfiguration overhead. The **image smoothing (FIR)** phase is necessary to remove the noise from the image. The **image gradient**, computed by applying the filter function with a window-approach, is used to highlight regions with high spatial derivatives. Next, the intensity value image and the direction value image, are computed during the **non-maximum suppression** stage. At this point we obtain an image with approximate edges detected, which are often corrupted by the presence of false-edges. In order to delete these non-edges the gradient array is now further reduced by **hysteresis**. The most computationally expensive parts of the system are the *image smoothing filter (FIR)*, the *image gradient* and the *hysteresis*. We have first implemented these functions as IP-Cores in VHDL and they have been plugged into the self reconfigurable architecture, as the one proposed in Figure 2. The resulting system is composed of four modules. The distribution of the application functions into these modules is shown in Table 7. Module m_1 contains the static side, i.e., the PowerPC

Table 7: Modules partitioning.

Tasks	Application Functions	Occupied Slices	Percentage
m_1	Static Side, non-maximum suppression f_c	2662	54
m_2	image smoothing (FIR) f_a	245	4
m_3	gradient f_b	2168	44
m_4	hysteresis f_d	5343	108

core and all the interface infrastructures as well as the function f_c (non-maximum suppression). The other three tasks correspond to one IP core implemented with reconfigurable hardware. The resource requirements of these IP cores are shown in last two columns of Table 7. As a result, m_1 is extended to support both f_c (non-maximum suppression) and f_d (hysteresis). This move does not incur any additional overhead for the realization of module m_1. Module m_1 already contained one application function (f_c). Therefore, necessary computational resources (a PowerPC core) and communication components to correctly interface the software functions with the IP cores have already been created and accounted for. The newly added application function f_d will also use this existing infrastructure. With this solution, hardware reconfiguration has to be taken into account because the static portion of the architecture, m_1, along with the two IP-Cores, the FIR Filter, m_2, and the image gradient function, m_3, are not going to fit into the available reconfigurable hardware resources. In order to have an efficient implementation using partial dynamic reconfiguration, we have to process enough data to justify the reconfiguration between the FIR and image gradient cores ($368ms$). This example has been proposed just to present a complete and real application that can be implemented using a self dynamic reconfigurable architecture, obviously the same application can be implemented using a bigger FPGA without needing any reconfiguration. Reconfigurable SoCs are particularly powerful platforms for image and video processing and other multimedia applications. These domains provide essential services for many emerging embedded systems i.e. Smart-Transportation and Biomedical architecture.

5. CONCLUSIONS AND FUTURE WORK

Preliminary results show that the Shining methodology, provides an effective and low cost approach to the partial dynamic reconfiguration and mixed HW-SW execution problems. Its strength lies both on the introduction of the partial dynamic reconfiguration degree of freedom at *design time*, and on the use of widely available tools. The proposed flow, called Shining, organizes the input specification into three different components: hardware, reconfigurable hardware and software, managed by proper portion of the methodology. A fully automated version of the entire flow is currently under development in order to provide a completely automatic management of problems such as task partitioning, and core creation, which are now only semi-automated.

6. REFERENCES

[1] Wayne Wolf. A decade of hardware/software codesign. In *Computer*, pages 38–43. IEEE Computer Society, 2003.

[2] Alberto Donato, Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. Coperating system support for dynamically reconfigurable soc architectures. In *IEEE-SOCC*, 2005.

[3] Mihai Budiu Srihari Cadambi Matt Moe R. Reed Taylor Seth Copen Goldstein, Herman Schmit. Piperench: A reconfigurable architecture and compiler. B. Monien and R. Feldmann, April.

[4] B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of fpgas. 1991.

[5] B. Blodget, S. McMillan, Brandon Blodget1, E. Keller P. J. Roxby, and P. Sundararajan1. A self-reconfiguring platform. *Field Programmable Logic and Applications*, 2003.

[6] Alberto Donato, Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. Exploiting partial dynamic reconfiguration for soc design of complex application on fpga platforms. In *13th IFIP International Conference on Very Large Scale Integration - IFIP VLSI-SOC*, pages 179–184, 2005.

[7] IBM corporation. *The CoreConnect Bus Architecture, white paper*. International Business Machines Corporation., 2004.

[8] Heiko Kalte, Mario Porrmann, and Ulrich Rückert. A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In *Proc. of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip*, 2002.

[9] Xilinx. *Embedded Development Kit EDK 8.1*. Xilinx, 2006.

[10] *Virtex-II Pro Data Sheet Virtex-II Pro^{TM} Platform FPGA Data Sheet*. Xilinx, 2003.

[11] Xilinx Inc. Two Flows of Partial Reconfiguration: Module Based or Difference Based. (XAPP290), November 2003.

[12] Xilinx Inc. *Early Access Partial Reconfiguration Guide*. Xilinx Inc., 2006.

[13] M. Porrmann H. Kalte, G. Lee and U. Rückert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *The 12th Reconfigurable Architectures Workshop (RAW 2005)*, 2005.

[14] Chao-Chee Ku and Ren-Kuan Liang. Accurate motion detection and sawtooth artifacts remove video processing engine for lcd tv. In *IEEE Transaction on Consumer Electronics*, volume 50, pages 1194–1201, November 2004.

[15] R. Maestra, F.J. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, December 2001.