Fast Integration of Hardware Accelerators for Dynamically Reconfigurable Architecture

Clément Foucher, Fabrice Muller, Alain Giulieri

Fast Integration of Hardware Accelerators for Dynamically Reconfigurable Architecture

Clément Foucher, Fabrice Muller, and Alain Giulieri
University of Nice-Sophia Antipolis,
Laboratoire d'Électronique, Antennes et Télécommunications (LEAT)/CNRS

{Clement.Foucher, Fabrice.Muller, Alain.Giulieri}@unice.fr

I. Introduction

Reconfigurable hardware offers an alternative to static hardwired Intellectual Property blocks (IPs) found in Application-Specific Integrated Circuits (ASICs). Field-Programmable Gate Arrays (FPGAs) can be used to bring reconfigurable hardware resources in designs. FPGA are seen as IPs that can adapt various behaviors depending on the needs. Reconfigurable hardware then makes applications taking advantage of hardware more flexible and even self-adaptive.

Their naturally dynamic nature also offers a way to create static designs. Indeed, implementing circuitry on a FPGA instead of an ASIC enables hardware to be updated even after a product release by distributing the new bitstream. They also offer prototyping capabilities, allowing a design to be tested without the need of silicon hard printing.

But the most exciting strength of reprogrammable hardware lies in its dynamic nature, offering software-like flexibility. Moreover, an aspect of reconfigurable hardware, Partial Dynamic Reconfiguration (PDR) is enhancing these perspectives. PDR enables part of a reconfigurable device to be changed on the fly without affecting the remaining part, continuing to run during the reconfiguration.

Without PDR, reconfigurable devices must be reconfigured at once, leading to consider the whole device as monolithic. For FPGA-hosted multi-IP designs, it is then impossible to change only one IP at a time, and the reconfiguration process suspended the whole design run. Reconfigurable devices, which used to be considered somehow as *offline reconfigurable* are now able to change their behavior online using PDR. This leads reconfigurable devices to enter a new stage, because the dynamic nature of hardware can be taken into consideration at runtime even for designs with multiple IPs.

However, the partial reconfiguration process is still in its early stages. The complexity of partial reconfiguration management and the need to adapt existing IPs are major issues in system designs. In some cases, this leads system designers to neglect the use of dynamic hardware in favor of more software resources, or static hardware using ASICs.

We developed a platform called the Simple Parallel platform for Reconfigurable Environment (SPoRE), which provides automatic management of PDR. SPoRE offers a way to integrate static legacy IPs in dynamic applications without needed adaptation, simply by describing the IP interface using generic syntax. Moreover, SPoRE enhances the prototyping capabilities of FPGAs as it allows fast parallel deploiements, which can be used to process testbenches on multiple FPGAs simultaneously. The SPoRE platform was originally presented in [1], in which we describe the SPoRE theoretical platform and implementations of this platform.

In our methodology, an application is designed as a set of computing elements implementing algorithms, called *kernels*. Applications are then regarded as dataflow graphs made of virtual kernels linked by data. Then, independently from the application, the kernels are given one or various implementation(s). Implementation choice, IPs reconfiguration, and communication between kernels are handled automatically by a runtime manager. Applications can also be automatically distributed on various execution nodes linked by a network.

In this article, we go in detail in the IP interface we developed. We present the IP interface description generic syntax enabling the kernel virtualization. Also, we present the IP encapsulation principle allowing the automatic interface handling process. We then present the results conducted on a sample application running on SPoRE.

First, section II presents related work already conducted in interface handling and reconfigurable platforms. Section III outlines the SPoRE applications architecture allowing kernel virtualization. Then, section IV goes in detail the interface handling mechanism. Afterwards, section V explains how we integrate legacy hardware IPs in SPoRE. Finally, section VI presents a sample application building process and results of its execution on a SPoRE implementation. We finish by a conclusion including development perspectives in section VII.

II. Related work

In this section, we take a look at related work in the field of hardware/software automatic interface handling, as well as reconfigurable platforms.

The most common way to interface hardware IPs in a system is to plug them on a bus. Buses can adapt various topologies depending notably on bandwidth requirements. Lately, a new kind of buses has emerged for designs with high number of IPs: the Network-on-Chip (NoC) paradigm, implementing real networks with routing and message passing directly on a chip [2].

Standard buses [3] use a register description to handle IPs access, with a master accessing a slave's memory mapping. The bus low-level protocol is generally standard, allowing transaction-based accesses through wrappers avoiding dealing directly with the bus signals. However, the master has to be aware of slave protocol itself, knowing which transactions must be done to obtain the expected result. For software-centered platforms, this is generally done using a driver that transforms high-level calls in bus-level transactions.

Using NoCs, the low-level protocol can match a standard message-passing protocol. For example, it is possible to implement NoC that supports the Message Passing Interface (MPI) [4], thus enabling MPI-aware IPs to directly use the standard protocol. But this kind of NoC is still unconventional and at this time, only existing in specific systems.

In static systems, in which the hardware does not change, the driver use is still relevant. But reconfigurable systems add application-dependent hardware, which is instantiated on reconfigurable resources depending on the application needs. Thus, applications must be aware of the specific reconfigurable IPs interface. Applications are then deeply dependent on specific IPs, which can restrict IPs re-use. The IP interface handling thus needs a higher level standardization in order to allow its use in reconfigurable systems.

A bus virtualization initiative is Open Core Protocol (OCP) [5], which provides a bus-independent protocol allowing IPs communication. It specifies higher-level interface specification than a bus that can be built around any kind of bus using a wrapper interface that will transform OCP transactions in bus-specific transactions. OCP defines standard read/write operations, including burst that allows reduced latency for transactions on multiple data words. It also adds broadcast operations that can be used to allow interactions between more than two IPs in a single transaction. Moreover, locked read-and-write are supported to allow mutual exclusion operations.

OCP-compatible IPs then communicate between them using the virtual OCP bus, allowing IP re-use between OCP designs. This adds interesting portability capabilities, but do not allow direct re-use of legacy IPs. Moreover, this protocol is still low level, thinking in terms of transactions that require the knowledge of the remote IP functioning.

Another standardization initiative is OpenFPGA CoreLib [6]. CoreLib aims at defining standard interfaces for hardware IPs interaction in order to guarantee interoperability, even for IPs not using a bus. This aims at enhancing internal FPGAs routing capabilities, using which point-to-point communication performances can supersede a bus's one.

CoreLib defines three interfaces characteristics: structural, temporal and control. Structural interface defines the signals used and their category: clock, reset, control and data. Moreover, this interface defines the nature of the signal: floating point value, integer, etc. Temporal interface is used to characterize signals timing: pipeline description, interval between control and data, etc. Finally, control interface describes the IP's synchronization with the rest of the system, e.g. how to start the IP and how to know work is done.

The combination of these three interfaces allows standard interaction between two hardware IPs, without having to know the details of their functioning. All these interfaces are described using Spirit Consortium IP-XACT standard [7], an XML-based standard for describing IPs interfaces. IP-XACT interface description can be automatically generated, thus providing direct interface specifications without needing a manual investigation.

Nevertheless, CoreLib is hardware oriented, and would be difficult to adapt in heterogeneous architectures such as High Performance Reconfigurable Computing, that are software-centered.

These two examples are interesting from an interface point of view, but lacks high level support for heterogeneous applications involving software units. We now focus on complete environments providing interfaces between hardware and software kernels.

Of notable interest is the Berkeley Operating system for ReProgrammable Hardware (BORPH) [8], which integrates the reconfigurable hardware accelerators as standard tasks in a Linux system. This is more than a simple interface, rather a complete Linux-based operating system. BORPH extends the software processes UNIX interface standard to hardware processes. This allows handling hardware processes the same way as software one, using the standard I/Os. As an example, BORPH makes possible using a pipe in a console between a hardware process and a software one.

The generalization of UNIX interface to hardware IPs allows using them in a tried and tested standard environment. It notably allows to easily change a process implementation from software to reconfigurable hardware without having to change the whole application. However, the specific protocol used to communicate with hardware tasks involves a dedicated conception of the IPs targeting BORPH, then impacting IP re-usability.

RAMPSoCVM [9] provides an automated management of reconfigurable resources with MPI access. Built on reconfigurable resources, the platform allows instantiating various software units, linking them using a reconfigurable NoC. A central unit is used as a server and distributes work to clients, serving as a virtualization layer hiding the actual platform implementation to the software application. All units have a bootloader that is able to load MPI applications and a MPI layer interfacing with the NoC. This approach proposes a standard interface for reconfigurable platforms, but would require a specific MPI interface to include hardware accelerators.

III. Application description

In SPoRE, a single kernel, carrying out a specific computation, can have multiple implementations, both hardware and software. Different hardware implementations can then provide different algorithms and levels of parallelization resulting in different performance versus area ratios. Moreover, a single hardware kernel can then be synthesized targeting various hardware technologies to allow its use on various platforms. The hardware/software capability of the platform ensures

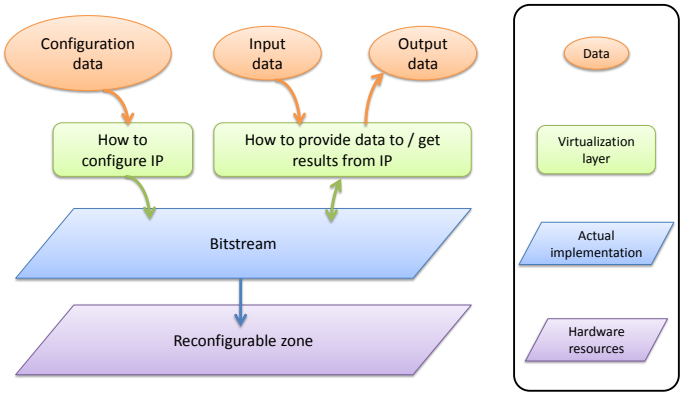

Fig. 1. Virtualization layer between data to process and actual hardware implementation.

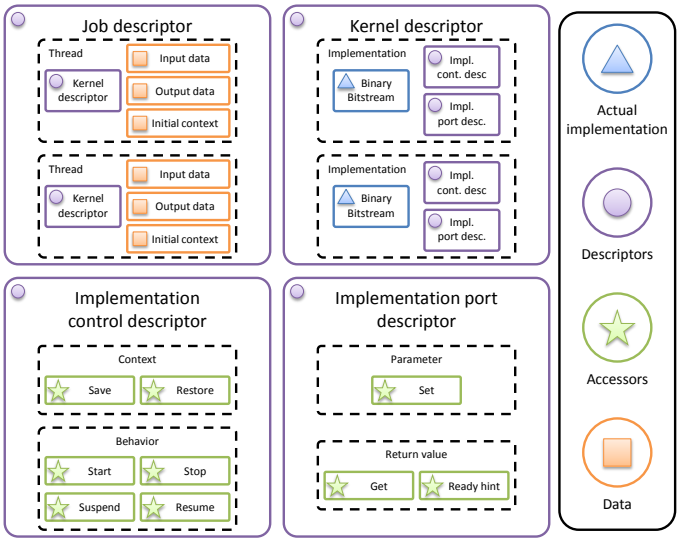

Fig. 2. Application structure hierarchy.

a kernel will always be schedulable, even when hardware resources are exhausted. The same as hardware, it is also possible to provide different binaries using different instruction sets for software kernels in order to allow portability.

In order to allow providing multiple implementations for a single kernel, we introduce a virtualization layer between the virtual kernel and the actual kernel implementation interface. This means a kernel will always be handled the same way, regardless of its actual implementation. Figure 1 illustrates the principle on a hardware implementation. On one side, the application provides references to data. Data includes both configuration used to set the kernel's context and the input data that has to be processed by the kernel. On the other side, a kernel implementation provides a bitstream or a binary that is able to carry out the required computations on the targeted platform. The virtualization layer then provides the required information on how to adapt data to this specific kernel, using what we call *accessors*.

The general structure of an application is made of a hierarchy of *descriptor* files, as presented on figure 2. At the top level, the *job descriptor* describes a list of threads to execute, i.e. kernels along with their input and output data. The general application is then represented by a job descriptor, referencing the virtual kernels to execute on the platform. If the platform is made of multiple nodes, a global scheduler will split the application in various job descriptors that will be dispatched on the nodes as sub-applications. The job descriptor contains references to *kernels descriptors*, as well as data files for input and output data, and a context representing the kernel configuration. At this point, there is no notion of the kernels implementations, and the application is then totally portable. A thread ability to be scheduled is done by looking at it input data: if the whole required data exists, the thread can be launched.

The lower level is made of the kernels themselves. A kernel is represented by a kernel descriptor, which contains the list of the different available implementations. Each implementation provides an actual mean to process computations: reference to a bitstream or a binary file. On each platform, or each node for a multi-node platform, all implementations may not be available due to the reconfigurable hardware technology and/or software instruction set.

With each implementation comes two descriptors: the *implementation control descriptor* and the *implementation port descriptor*. These are these two files that contain the accessors, describing how to interact with the implementation.

IV. Accessors

The accessors contained in both implementation control and port descriptors implement the actual kernel implementation virtualization. Indeed, a single data file, e.g. containing a kernel initial configuration, can match any implementation of a kernel, the corresponding accessor describing how to feed the implementation with this data.

The implementation control descriptor contains the accessors that define the base kernel control transactions, such as starting and stopping the IP, or saving and restoring the context for preemption-capable IPs. We call input data *parameters* and output data *return values*, as an analogy with software procedures. The port descriptor contains the dataflow accessors, indicating for each parameter how to feed the IP, and for return values how to check if the data is ready and retrieve it.

For hardware kernels, accessors structure contains a sequence of bus-based actions to execute over an IP. Each action has a direction and a type. Direction can be either read or write, while type can be single action over register, FIFO action over register, action over memory range or pointer type action. For write action, the value can be provided as constant (e.g. "Write 0x80000000") or as reference to a data file (e.g. "Write value #X from data file #Y"). Data size and count is provided (e.g. "10 elements of 4 bytes") to indicate the data characteristics. Finally, an offset indicates the address of the register on which to proceed to the action.

Software kernels do the same, except that instead of registers, we represent parameters. The offset is then replaced

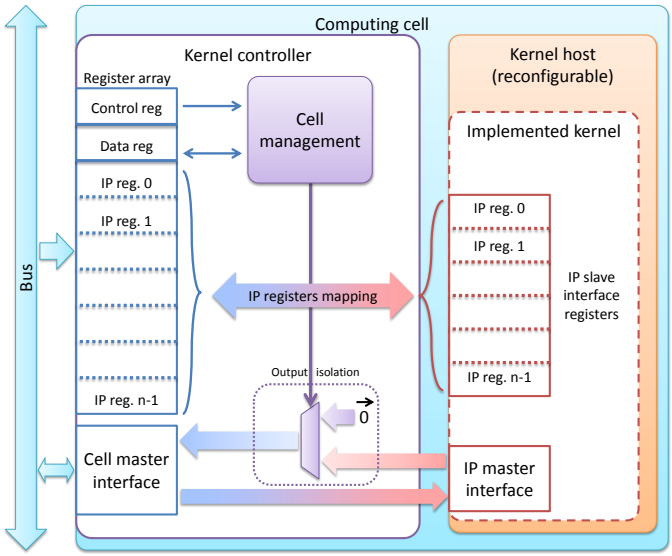

Fig. 3. Hardware IP wrapper.

by the parameter number. Moreover, software kernels support string parameters, while hardware kernels only have values. Software implementation control descriptor also makes the distinction between binaries belonging to the environment or provided specifically for the application. It is then possible to call system executables like in a console, by providing their path.

Thus, by indicating a sequence of such actions, an accessor provides a complete step-by-step procedure to carry on for a particular interaction. For hardware IPs, the ones having a sole slave interface will require providing the data directly to the IP registers. But if a master interface is available, pointer actions allows interacting directly with memory. Pointer parameters will be executed on two steps: interaction with memory and interaction with IP. Interaction with memory consists in reserving enough memory space to contain the data, and fill it with actual data if this is an input parameter. Then, the pointer is given to the IP as an address, and data size if needed. Using pointer actions, the dataflow between two kernels can be done without reading or writing to the memory, simply giving the source IP's output pointer to the sink IP. Software kernels only support pointer parameters for now, as we did not implement signal handling.

V. Hardware IP wrapping

We want to make possible using legacy IPs, possibly static, i.e. not designed for a reconfigurable environment, the only requirement being that there must be bus-based. For dataflow IPs not disposing of a bus interface, it will still be possible to add a wrapping interface adapting registers actions to the actual IP signals.

We designed a wrapping interface taking place between the reconfigurable core and the bus, as shown on figure 3. We call this element the *kernel controller*. This interface is notably in charge of signal isolation during reconfiguration process. Indeed, while reconfiguring an IP, random signals can appear on its outputs. This can be a problem if the IP has a master interface, generating uncontrolled commands on the bus, possibly causing unexpected requests or even causing the bus arbiter to crash. This is solved by a multiplexer forcing output signals to '0' while reconfiguring.

Another role of the kernel manager is to include monitoring elements. We included a timing monitor, able to measure the execution time of the IP. This is used to obtain exact (cycle accurate) timings for the IPs. In future, we can imagine using these timings as a result to feed the scheduling process and adapt the policy, thus enabling self-adaptive architectures.

To command the kernel controller, we have two registers: a control register and a data register. The control register is used to send a command to the kernel controller, followed by a read or a write to the data register. Also, a read in the control register indicates the characteristics and configuration of the kernel controller: timer availability, kernel host connection state, etc. Thus, the *computing cell*, that is the couple kernel controller/kernel host, has a register array consisting in these two registers followed by the dynamic IP's registers. The register address translation is automatically handled by the runtime environment, and do not need to be taken in consideration in accessors writing.

VI. SPoRE implementation and results

We made a SPoRE node implementation using Xilinx's ML507 development board [1]. ML507 is built around Virtex 5 fx70t FPGA. The PowerPC 440 integrated in the FPGA is used to run a Linux embedded system, in charge of the node's management. We use a 2.6.34 embedded Linux available from Xilinx [10]. Using a standard Linux system allows network communications using simple socket procedures. The Linux-based system can make decision on thread scheduling and hardware resources reconfiguration, using the SPoRE runtime environment. The runtime environment is also in charge of multi-node communication and hardware resources reconfiguration.

SPoRE network is composed of a data server and a global scheduler additionally to the computing nodes. The global scheduler is in charge of dispatching jobs over the boards. Using the job description, the nodes then contact the data server to download the descriptors and the bitsteams/binaries needed for the run.

The kernel's computation can be run on the PowerPC device (running at 400 MHz) or using the reconfigurable computing cells (running at 100 MHz), which interface is accessible to the runtime environment using a Linux driver. This standard driver allows cells interaction using the operations allowed in the descriptors.

A specific portion of the RAM is reserved to the computing cells for storing input and output data. This enables cell-to-cell data transmission with minimal runtime environment interaction, transmitting only the data address and size from one IP to another.

TABLE I
EXECUTION TIMES OF AES ENCRYPT AND DECRYPT APPLICATIONS.

Data size		512 KiB	1 MiB	5 MiB
Software	encoder execution	426,580 $\mu s \pm 0.37\%$	717,269 $\mu s \pm 0.63\%$	3,049,773 $\mu s \pm 0.83\%$
	decoder execution	550,588 $\mu s \pm 0.87\%$	972,432 $\mu s \pm 0.23\%$	4,349,304 $\mu s \pm 0.63\%$
Hardware (1 cell)	encoder execution	52,926 $\mu s \pm 0.03\%$	105,847 $\mu s \pm 0.02\%$	529,166 $\mu s \pm 0.01\%$
	decoder execution	52,942 $\mu s \pm 0.06\%$	105,853 $\mu s \pm 0.01\%$	529,178 $\mu s \pm 0.01\%$
	encoder reconfiguration	825 $\mu s \pm 0.67\%$		
	decoder reconfiguration	906 $\mu s \pm 0.12\%$		
	encoder total	53,751 $\mu s \pm 0.03\%$	106,673 $\mu s \pm 0.02\%$	529,991 $\mu s \pm 0.01\%$
	decoder total	53,848 $\mu s \pm 0.06\%$	106,759 $\mu s \pm 0.01\%$	530,084 $\mu s \pm 0.01\%$
Hardware (2 cells)	encoder execution	28,541 $\mu s \pm 0.15\%$	57,974 $\mu s \pm 0.08\%$	285,656 $\mu s \pm 0.08\%$
	decoder execution	26,859 $\mu s \pm 0.03\%$	54,092 $\mu s \pm 0.04\%$	271,887 $\mu s \pm 0.05\%$
	encoder reconfiguration	1,671 $\mu s \pm 0.46\%$		
	decoder reconfiguration	1,820 $\mu s \pm 0.21\%$		
	encoder total	30,177 $\mu s \pm 0.15\%$	59,642 $\mu s \pm 0.09\%$	287,329 $\mu s \pm 0.08\%$
	decoder total	28,670 $\mu s \pm 0.03\%$	55,912 $\mu s \pm 0.04\%$	273,706 $\mu s \pm 0.05\%$

The descriptors are written using XML, which can be easily written and parsed using widely available libraries. We use libxml2 [11], a free and easy XML handling library that allows both file parsing and writing.

In order to handle hardware reconfiguration, we rely on the Fast Reconfiguration Manager (FaRM) [12]. FaRM is an IP that wraps the Internal Configuration Access Port (ICAP) [13], the reconfiguration controller located inside the FPGA. FaRM allows direct reconfiguration just by using a partial bitstream previously stored in the memory. Moreover, it supports bistream compression that reduces transfers overheads.

In order to test the interface handling validity, we built a sample application with kernels having both a software and a hardware application. This application consists in an AES data encryption followed by its decrypting.

For the software implementation, we rely on the OpenSSL library [14]. OpenSSL is a cryptographic library that makes available crypting and decrypting routines, which can be called in an application. OpenSSL can also be built as a standalone application that we can call using command-line. This is the second solution we use, as SPoRE nodes' Linux system already includes this library.

The hardware kernels cores are taken from the OpenCores online IP library [15] that makes available lots of freely available IPs. OpenCores proposes an AES encrypting and decrypting core proceeding on 128-bits words. Its interface is very simple:

- Data signals
  - Encryption key
  - Input data
  - Output data
- Control signals
  - Ability to choose between encryption and decryption
  - Load key
  - Start computations
  - Indication that computations are over

This interface is not based on addressable registers and thus cannot be immediately connected to a bus. In order to make this IP bus-compliant and allow it to interact directly with data stored in memory, we wrapped the IP in a tiny additional interface. This wrapper includes a light DMA to allow direct memory data fetching, and links the IP signals to bus-accessible registers. We also made the selection between encryption and decryption static, resulting in two different IPs, an AES encrypter and an AES decrypter.

The application was run on a SPoRE node containing two computing cells. Four different partial bitstreams are generated, in order to dispose of both AES encryption and decryption IPs on both cells.

Then, we build the application components using our XML-based structure. We require kernel descriptors for both encrypting and decrypting IPs, each containing both the software and hardware implementations description. To build the implementations, we describe the IPs accesses methods using the accessors. Finally, the job descriptor is built to indicate which kernels should be called and their data links. We present the hardware implementation control descriptor on figure 4 and the software one on figure 5.

The whole application XML description is less than 6 KiB, and needed only a few hours to write it. We need to add the sum of the bitstreams size to the application weight, which is 502 KiB. The OpenSSL library size is not counted as the library already belongs to our system, and do not need to be carried by the application. We also have to add the AES hardware IPs wrapping time, which took about one day.

Then, we proceeded to various runs of the application, using different configurations. The SPoRE runtime allows monitoring execution times of the both hardware and software kernels, as well as reconfiguration time for hardware ones. The application monitoring was done using the following configurations:

- Data size: 512 KiB, 1 MiB and 5 MiB
- Kernels nature: software, hardware using one cell and hardware using two cells

Running the hardware kernels on *n* cells means the *m* bytes of data are equally distributed into $\frac{m}{n}$ bytes to the cells. This allows more parallelism in the execution of the application, which is a strength of using a hardware implementation for a

```
<Implementation_control_descriptor ID="19" Name="Hw AES enc. and dec. ICD">
  <Context>
    <Restore>
      <!--Cypher key-->
      <Action Type="Memory" Direction="Write" Data_reference="0"
              Offset="36" Size="4" Count="4"/>
    </Restore>
  </Context>
  <Behavior>
    <Start>
      <Action Type="Register" Direction="Write" Data_constant="01"
              Offset="20" Size="4"/>
    </Start>
  </Behavior>
</Implementation_control_descriptor>
```

Fig. 4. Hardware implementation control descriptor (ICD).

```
<Implementation_control_descriptor ID="20" Name="Software AES encoder ICD">
  <Context>
    <Restore>
      <Action Type="String" Prefix="enc" Data_ignored="" Offset="1"/>
      <Action Type="String" Prefix="-e" Data_ignored=""/>
      <Action Type="String" Prefix="-aes-128-ecb" Data_ignored=""/>
      <Action Type="String" Prefix="-nosalt" Data_ignored=""/>
      <Action Type="String" Prefix="-nopad" Data_ignored=""/>
      <Action Type="String" Prefix="-K" Data_ignored="" Offset="2"/>
      <!--Cypher key-->
      <Action Type="String" Data_reference="4" Offset="3"/>
    </Restore>
  </Context>
</Implementation_control_descriptor>
```

Fig. 5. Software implementation control descriptor (ICD).

kernel. The execution and reconfiguration times are displayed in table I. Each application configuration has been run ten times, giving an average time and a variation around this value.

Unlike software, hardware total run times are not only the execution times, but needs to add the IP reconfiguration time, adding a constant overhead to the variable execution time, which depends on data size. The conclusions we can extract from these measures are presented in table II.

We can state that using a one-cell hardware implementation allows a 85-89% gain on software execution time, and 92-94% with a two-cell implementation. The hardware implementation represents then a great time gain compared to the software one on this embedded platform.

The reconfiguration overhead varies between 0.2% and 1.6% of the execution time for the one-cell configuration, and between 0.6% and 6.3% with two cells. Except for the 512 KiB configuration with two cells, in which the reconfiguration overhead is over 5%, we can state that the time is negligible. But this case is also the one with the most benefit compared to the software kernels implementation.

VII. Conclusion and Future Work

In this paper, we presented an interface description to automatically handle reconfigurable hardware IPs in an application. Our aim was to define a bus-based interface which could allow hosting reconfigurable hardware IPs in an application. The interface handling principle, based on the IP interface

TABLE II
Conclusions on the execution time measures.

Data size		512 KiB	1 MiB	5 MiB
One cell	Speedup versus software	88.99%	87.37%	85.67%
	Reconfiguration overhead	1.64%	0.82%	0.16%
Two cells	Speedup versus software	93.98%	93.16%	92.42%
	Reconfiguration overhead	6.3%	3.12%	0.63%

virtualisation allows all bus-based transactions to be described using a specific syntax, for example in XML. Based on that, we were able to define *accessors*, which are bus transaction sequences used to perform specific actions on IPs such as configuring or starting it.

Combined to our descriptor-based application description, we were able to conceive a sample application involving AES to demonstrate the easy integration process of legacy IPs. Moreover, this application validated the virtual kernel principle, allowing us to define implementation-independent kernels. We were then able to provide both hardware and software implementation of our AES kernels, by discarding the actual kernel nature at application building. As SPoRE natively includes monitoring tools, we were able to extract execution times with detail on reconfiguration overhead.

We conducted measures to compare hardware and software kernels execution times, and obviously found the hardware kernels were faster than the software implementations on our test platform, as expected. Furthermore, we shown that in most case, the overhead introduced by the reconfiguration time was negligible, all the more when there are more data.

About platform hardware support, we will integrate a master controller in the kernel controller in order to allow slave-only cores to directly use pointer data transfers. Indeed, direct pointer support for data management allows IPs to be quite totally independent from the central node management, decreasing IPs interface interaction needs and thus IP management costs.

On the general platform description, we are interested in IP-XACT signal description, as using a standard is always a best solution than building a specific syntax. We are investigating IP-XACT compliance with our accessors syntax in order to automatically generate IPs interface. This automatization would reduce work on writing the hardware kernels implementations descriptors, as the accessors description step could rely on these automatically generated files. Moreover, using an IP-XACT description of IPs interface that are not bus-based could allow an automatic generation of the bus wrapper, widening the number of IPs compatible with SPoRE.

We still need to integrate support for communication between IPs in port descriptor, in order to allow other kernel relationship than data dependency. We are notably interested in supporting MPI most common calls to enhance compatibility with existing applications.

Finally, we are looking into using the SPoRE monitoring tools informations to enhance the scheduling capability. Combined to our multi-implementation support for kernels, the

execution timing results of a kernel run can help the scheduler to make better implementation choices. This would allow really self-adaptive applications, taking the actual platform resources into consideration at run time.

REFERENCES

[1] C. Foucher, F. Muller, and A. Giulieri, "Online code-sign on reconfigurable platform for parallel computing," *Microprocess. Microsyst., in press*, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2011.12.007

[2] L. Benini and G. D. Micheli, "Networks on Chips: A new SoC paradigm," *Computer*, vol. 35, pp. 70–78, 2002. [Online]. Available: http://dx.doi.org/10.1109/2.976921

[3] Power.org, "Embedded bus architecture report," http://www.power.org/resources/downloads/Embedded_Bus_Arch_Report_1.0.pdf, Tech. Rep., April 2008.

[4] R. Hempel and D. W. Walker, "The emergence of the MPI message passing standard for parallel computing," *Comput. Stand. Interfaces*, vol. 21, pp. 51–62, May 1999. [Online]. Available: http://dx.doi.org/10.1016/S0920-5489(99)00004-5

[5] OCP-IP, *Open Core Protocole Specification*, http://ocpip.org, 2001.

[6] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "OpenFPGA CoreLib core library interoperability effort," *Parallel Comput.*, vol. 34, no. 4-5, pp. 231 – 244, 2008, reconfigurable Systems Summer Institute 2007. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2008.03.004

[7] S. S. W. G. Membership, *IP-XACT User Guide v1.2*, http://www.spiritconsortium.org, July 2006.

[8] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 14:1–14:28, January 2008. [Online]. Available: http://dx.doi.org/10.1145/1331331.1331338

[9] D. Gohringer, S. Werner, M. Hubner, and J. Becker, "Rampsocvm: Runtime support and hardware virtualization for a runtime adaptive mpsoc," in *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, ser. FPL '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–184. [Online]. Available: http://dx.doi.org/10.1109/FPL.2011.41

[10] Xilinx open source resources, http://xilinx.wikidot.com, 2011.

[11] Libxml2 XLM parsing library, http://www.xmlsoft.org.

[12] F. Duhem, F. Muller, and P. Lorenzini, "FaRM: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, Eds. Springer Berlin / Heidelberg, 2011, vol. 6578, pp. 253–260. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19475-7_26

[13] Xilinx, Inc., *Partial Reconfiguration User Guide*, http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/ug702.pdf, October 2010.

[14] OpenSSL cryptographic library, http://www.openssl.org.

[15] Opencores open source hardware IP-cores library, http://opencores.org.